SCIENCES SUP

Cours et exercices corrigés

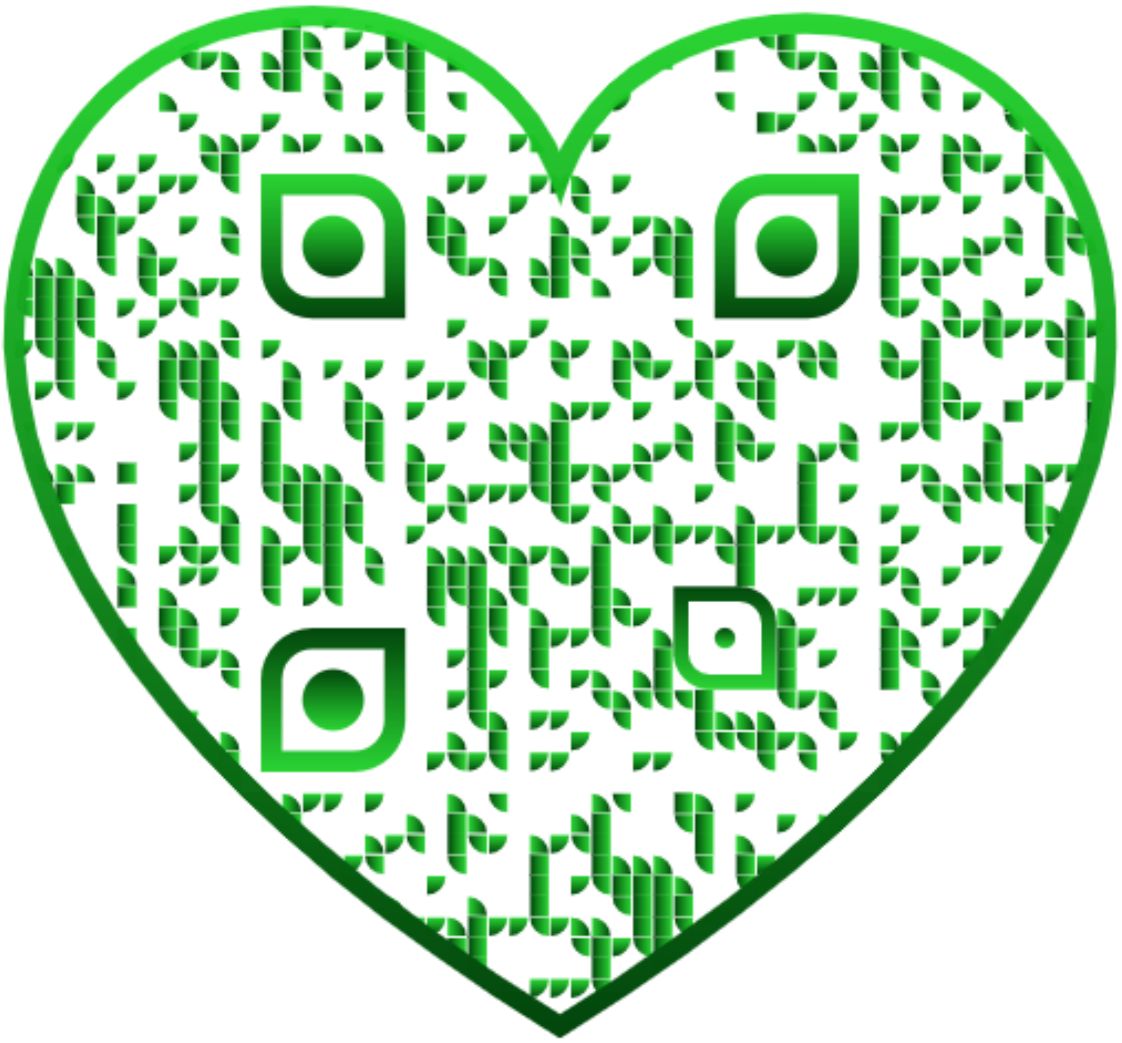
IUT • BTS • Licence • Écoles d'ingénieurs • Formation continue

PROGRAMMATION EN C++ ET GÉNIE LOGICIEL

COMPLÉMENTS en ligne

Vincent T'kindt

DUNOD



PROGRAMMATION EN C++ ET GÉNIE LOGICIEL

Génie logiciel
David Gustafson
208 pages
Schaum's, EdiScience, 2003

Structures de données avancées avec la STL
Programmation orientée objet en C++
Philippe Gabrini
432 pages
Dunod, 2005

PROGRAMMATION EN C++ ET GÉNIE LOGICIEL

Cours et exercices corrigés

Vincent T'kindt
Maître de conférences au département Informatique
de l'École Polytechnique de l'université François Rabelais de Tours

DUNOD

Le pictogramme qui figure ci-contre mérite une explication. Son objet est d'alerter le lecteur sur la menace que représente pour l'avenir de l'écrit, particulièrement dans le domaine de l'édition technique et universitaire, le développement massif du photocopillage.
Le Code de la propriété intellectuelle du 1er juillet 1992 interdit en effet expressément la photocopie à usage collectif sans autorisation des ayants droit. Or, cette pratique s'est généralisée dans les établissements d'enseignement supérieur, provoquant une baisse brutale des achats de livres et de revues, au point que la possibilité même pour les auteurs de créer des œuvres nouvelles et de les faire éditer correctement est aujourd'hui menacée.
Nous rappelons donc que toute reproduction, partielle ou totale, de la présente publication est interdite sans autorisation de l'auteur, de son éditeur ou du Centre français d'exploitation du droit de copie (CFC, 20, rue des Grands-Augustins, 75006 Paris).

© Dunod, Paris, 2007
ISBN 978-2-10-050634-7

Le Code de la propriété intellectuelle n'autorisant, aux termes de l'article L. 122-5, 2° et 3° a), d'une part, que les « copies ou reproductions strictement réservées à l'usage privé du copiste et non destinées à une utilisation collective » et, d'autre part, que les analyses et les courtes citations dans un but d'exemple et d'illustration, « toute représentation ou reproduction intégrale ou partielle faite sans le consentement de l'auteur ou de ses ayants droit ou ayants cause est illicite » (art. L. 122-4).
Cette représentation ou reproduction, par quelque procédé que ce soit, constituerait donc une contrefaçon sanctionnée par les articles L. 335-2 et suivants du Code de la propriété intellectuelle.

Table des matières

Avant-propos XI

CHAPITRE 1 • **NOTIONS DE BASE SUR LE GÉNIE LOGICIEL** 1

1.1 Qu'est-ce que le génie logiciel ? 1

1.2 Le cycle de vie d'un logiciel 2

1.3 Spécification et conception d'un logiciel 4

1.3.1. Les commentaires 5

1.3.2. Les exceptions 5

1.3.3. La spécification logique d'une fonction 8

1.3.4. Une première vision des classes 11

1.4 Quelques règles de bonne programmation 11

1.4.1. Règles liées à la spécification du programme 12

1.4.2. Règles liées à la conception du programme 18

CHAPITRE 2 • **GÉNÉRALITÉS SUR LE LANGAGE C++** 23

2.1 Mots-clefs, instructions et commentaires 23

2.2 La gestion des variables en langage C++ 24

2.2.1. Déclaration et initialisation des variables 25

2.2.2. Portée et visibilité des variables 26

2.3 Notion de référence 28

2.3.1. Passage d'arguments par référence à une fonction 28

2.3.2. Les variables références 30

2.4 L'en-tête d'une fonction 31

2.5 Éviter les problèmes d'inclusions multiples d'interfaces 31

CHAPITRE 3 • **LES OBJETS** 35

3.1 Le retour des structures 35

3.1.1. Déclarer et définir des structures 35

3.1.2. Utiliser des structures 37

3.2 Les classes 38
3.2.1. Déclarer et définir des classes 38
3.2.2. Utiliser des classes 41
3.2.3. Affecter un objet d'une classe dans un autre objet de la même classe : enjeux et dangers 42
3.3 Variables et attributs statiques 46
3.3.1. Variables statiques n'appartenant pas à une classe 46
3.3.2. Attributs statiques 47
3.4 Constructeurs et destructeur 49
3.4.1. Les constructeurs 49
3.4.2. Le destructeur 56
3.5 Gestion d'objets dynamiques : les opérateurs `new` et `delete` 57
3.5.1. Allocation d'un élément unique ou d'un tableau d'éléments 57
3.5.2. Désallocation d'un élément unique ou d'un tableau d'éléments 60
3.6 Tout sur la vie des objets : synthèse 60
3.6.1. Initialisation d'un objet dès sa déclaration 61
3.6.2. Différentes catégories d'objets 61
3.7 Comment bien écrire une classe ? 63

CHAPITRE 4 • **LES TRAITEMENTS** 65
4.1 Passage d'arguments par défaut 65
4.2 Échange d'objets entre fonctions 66
4.3 Propriétés des fonctions membres 67
4.3.1. Spécification `inline` 67
4.3.2. Méthodes statiques 70
4.3.3. Auto-référence d'une classe 71
4.3.4. Fonctions membres constantes 71
4.3.5. Pointeurs sur les membres d'une classe 73

CHAPITRE 5 • **LES FONCTIONS ET LES CLASSES AMIES** 75
5.1 Amis et faux amis : le point de vue du génie logiciel 75
5.2 Le cas d'une fonction amie d'une classe 76
5.3 Le cas d'une méthode d'une classe amie d'une autre classe 77
5.4 Toutes les fonctions membres d'une classe amies d'une autre classe 77

CHAPITRE 6 • **LES EXCEPTIONS** 79
6.1 Gestion des exceptions en langage C++ 79
6.1.1. Lever une exception 80
6.1.2. Attraper une exception 81
6.1.3. Quel gestionnaire d'exceptions choisir ? 83
6.1.4. Gestion hiérarchisée des exceptions 85
6.1.5. Liste d'exceptions valides 87
6.2 Fonctions liées au traitement des exceptions 88
6.2.1. Différentes façons d'arrêter un programme 88
6.2.2. Remplacement de fonctions pour le traitement des erreurs 89
6.3 Utilisation des exceptions dans le développement de programme 90
CHAPITRE 7 • **L'HÉRITAGE** 91
7.1 L'héritage simple 91
7.1.1. Mise en œuvre 92
7.1.2. Contrôles d'accès 93
7.1.3. Accès aux membres de la classe de base 95
7.1.4. Compatibilité avec la classe de base 96
7.1.5. Appels des constructeurs et des destructeurs 98
7.1.6. Cas du constructeur de recopie 100
7.2 L'héritage multiple 101
7.2.1. Mise en œuvre 101
7.2.2. Membres homonymes dans les classes mères 101
7.2.3. Constructeurs et destructeur 105
CHAPITRE 8 • **LA SURCHARGE** 107
8.1 La surcharge de fonctions et de méthodes 107
8.1.1. Cas des fonctions/méthodes à un argument 108
8.1.2. Différenciation des prototypes 110
8.1.3. Cas des fonctions/méthodes à plusieurs arguments 111
8.1.4. Surcharge de fonctions membres 114
8.2 La surcharge d'opérateurs 114
8.2.1. Mécanisme 115
8.2.2. Portée et limites 116
8.2.3. Opérateurs ++ et - - 117
8.2.4. Opérateur d'affectation = 118
8.2.5. Opérateur d'indexation [] 120

8.2.6. Opérateur d'accès aux membres d'une classe (->) 121
8.2.7. Opérateur de déréférencement (*) 123
8.2.8. Opérateurs `new` et `delete` 124
8.3 La surcharge de types 126
8.3.1. Rappels sur les conversions 126
8.3.2. Conversion d'une classe vers un type de base 128
8.3.3. Conversion d'un type de base vers une classe 129
8.3.4. Conversion d'une classe vers une autre classe 131
8.3.5. Exemples de conversions 132

CHAPITRE 9 • **LES PATRONS DE FONCTIONS ET DE CLASSES** 135
9.1 De la généricité avec les patrons ! 135
9.2 Les patrons de fonctions/méthodes 136
9.2.1. Création d'un patron de fonctions/méthodes 136
9.2.2. Instanciation et utilisation 136
9.2.3. Spécialisation d'un patron de fonctions/méthodes 139
9.2.4. Sucharger un patron de fonctions/méthodes 139
9.3 Les patrons de classes 140
9.3.1. Création d'un patron de classes 141
9.3.2. Les membres statiques au sein d'un patron 142
9.3.3. Instanciation et utilisation 143
9.3.4. Préciser des types par défaut pour les types génériques 144
9.3.5. L'amitié et les patrons 145
9.3.6. Spécialisation du patron 146
9.3.7. Aller encore plus loin avec les patrons 147

CHAPITRE 10 • **INTRODUCTION AU POLYMORPHISME ET AUX MÉTHODES VIRTUELLES** 149
10.1 Qu'est-ce que le polymorphisme ? 149
10.2 Les méthodes virtuelles 151
10.2.1. Définition et déclaration 151
10.2.2. Méthodes virtuelles et héritages complexes 153
10.2.3. Influence des contrôles d'accès 154
10.2.4. Le cas des valeurs par défaut 155
10.2.5. Les méthodes virtuelles pures et les classes abstraites 156

CHAPITRE 11 • **LES FLOTS** 159
11.1 Gérer vos entrées/sorties avec les flots 159
11.2 Classes d'entrées/sorties 161
11.2.1. Écrire à l'écran : le flot `cout` 162
11.2.2. Lire à l'écran : le flot `cin` 164
11.2.3. Manipuler des fichiers 165
11.3 Statuts d'erreur sur un flot 169
11.4 Formater vos flots 171

EXERCICES CORRIGÉS 175

ANNEXES 209

CHAPITRE A • **LES MOTS-CLEFS DU LANGAGE C++ ET LA CONSTRUCTION D'IDENTIFIANTS** 211

CHAPITRE B • **TYPES DE BASE DU LANGAGE C++** 213

CHAPITRE C • **PASSAGE D'ARGUMENTS À UNE FONCTION** 215
C.1 Le passage par valeur 215
C.2 Le passage par adresse 216

CHAPITRE D • **LISTE DES OPÉRATEURS SURCHARGEABLES** 219

CHAPITRE E • **LA CLASSE `CEXCEPTION`** 221

CHAPITRE F • **LA CLASSE `CLISTE`** 227

BIBLIOGRAPHIE 238

INDEX 239

Avant-propos

Le langage C++ est un langage de programmation, c'est-à-dire qu'il vous permet de créer des logiciels, ou programmes, exécutés par votre ordinateur. Il s'agit d'un langage très répandu et réputé pour sa puissance et sa flexibilité : on trouve des compilateurs C++ sur tous les types d'ordinateur, du micro-ordinateur de type PC ou Macintosh, à la station de travail professionnelle. Du système d'exploitation Unix en passant par le système d'exploitation Windows. Maîtriser le langage C++ est un atout indispensable dans le monde de l'informatique.

Cet ouvrage est dédié à l'apprentissage de la programmation en langage C++, ce qui recouvre à mon avis deux éléments essentiels. Le premier est l'apprentissage du langage en lui-même, c'est-à-dire les instructions et les règles qui correctement utilisées permettent de *faire faire quelque chose* à votre ordinateur. Le second élément important pour apprendre à programmer en langage C++ est lié à la façon d'écrire ces instructions pour limiter le nombre de *bugs*, pour permettre une lecture plus facile de votre programme, pour l'optimiser et vous permettre de le faire évoluer plus facilement ultérieurement... Dans le langage des informaticiens cela s'appelle faire du *génie logiciel*. Dans cet ouvrage vous trouverez non seulement, comme dans tout ouvrage sur le langage C++, tous les éléments du langage mais également un ensemble de recommandations recueillies dans le domaine du génie logiciel et qui vous guideront dans l'écriture de vos programmes. En langage C++ on vous explique comment créer des fonctions, ici vous verrez également des recommandations

sur le bon nombre de paramètres, sur ceux qui ne sont pas nécessaires, sur la façon de nommer vos fonctions, sur la façon de découper votre programme en classes...

Mais tout d'abord faisons un peu d'histoire pour bien appréhender la philosophie du ++. Le langage C++ fait partie des langages orientés objets, c'est-à-dire dont toute la programmation est centrée sur la notion d'objet. Il existe de nombreux langages orientés objets, plus ou moins récents, comme Smalltalk, Java... Par ailleurs, de nombreux ateliers de développement intègrent maintenant leur propre langage orienté objet. C'est le cas notamment de l'atelier WinDev et du W-langage. Tous ces langages reposent sur les mêmes principes généraux tels que l'héritage, l'encapsulation ..., mais tous ne sont pas conçus à l'identique : certains comme le Java ou Smalltalk sont des langages objets purs, et d'autres sont des extensions de langages existants. C'est le cas du langage C++ qui étend le langage C en y incluant les principes des langages orientés objets. Le langage C a été imaginé au début des années 1970, dans les laboratoires de recherche Bell de la compagnie américaine AT&T. Ses concepteurs, Brian Kernighan et Dennis Ritchie, l'ont conçu comme un langage de bas niveau, censé permettre au programmeur d'utiliser au mieux les ressources de son ordinateur. Initialement conçu pour le système Unix, le langage C s'est rapidement répandu sur tous les systèmes d'exploitation usuels. D'ailleurs, les systèmes Windows de Microsoft sont développés à l'aide du langage C principalement. Ce succès a conduit à une standardisation du langage de 1983 à 1988 par un comité de l'ANSI[1]. L'objectif de celui-ci était de clairement définir ce qu'était le langage C : ses instructions, ses règles et ses limites.

L'histoire du langage C++ débute dans le même laboratoire de recherche que le langage C. En 1979, Bjarne Stroustrup développe le langage *C with classes* qui est une extension basique du langage C incluant, entre autres, la définition de classes et l'héritage simple. Ce langage continu de s'enrichir et c'est en 1983 qu'il change de nom pour s'appeler tel que nous le connaissons aujourd'hui, le langage C++. De 1986 à 1989, l'utilisation du langage C++ se développe et différents compilateurs voient le jour. Comme dans le cas du langage C, un comité de l'ANSI a été chargé de 1989 à 1998 de normaliser le langage C++ (norme ISO/IEC 14882) de sorte que, quel que soit le compilateur utilisé, il reste le même. Bien sûr, cela n'empêche pas les différents compilateurs de permettre plus que la norme. Par ailleurs, le langage C++ reste en constante évolution pour intégrer les technologies les plus récentes.

Ce bref historique permet de bien comprendre ce qu'est le langage C++ : une extension du langage C. C'est pour cette raison que pour programmer en C++ il faut savoir programmer en C[2] : en apprenant le langage C++ vous apprendrez avant tout les nouvelles instructions et nouveaux concepts par rapport au langage C. Dans cet

1. American National Standard Institute.
2. Ou tout du moins avoir de bonnes notions du langage C.

ouvrage, nous verrons le langage C++ tel qu'il est présenté dans sa normalisation ANSI.

Alors, par où commencer ? Comment allons nous débuter votre apprentissage de la conception et la programmation en langage C++ ? Dans une première partie, constituée du chapitre 1, nous verrons les notions élémentaires du génie logiciel, ces notions étant illustrées ensuite dans le reste de cet ouvrage. Dans une deuxième partie, qui regroupe les chapitres 2, 3 et 4, nous aborderons les éléments de base du langage C++. À partir de là vous saurez créer des classes, y définir des attributs et des méthodes. La vie des objets n'aura plus aucun secret pour vous. La troisième partie, la plus importante en terme de taille, regroupe les chapitres 5 à 11 et présente des mécanismes particuliers du langage : l'amitié, les exceptions, l'héritage, la surcharge, les patrons, le polymorphisme et les flots. Enfin, vous trouverez dans une quatrième et dernière partie une collection d'exercices corrigés vous permettant de travailler les notions vues dans les chapitres précédants. Les exercices sont classés par chapitre et je vous conseille de faire ceux associés à un chapitre dès que vous en avez terminé la lecture.

Bienvenue dans l'univers du langage C++ !

Chapitre 1

Notions de base sur le génie logiciel

1.1 QU'EST-CE QUE LE GÉNIE LOGICIEL ?

Le génie logiciel a fait son apparition dans les années 1980 aux États-Unis suite à ce que l'on a appelé la *crise du logiciel*. À l'époque, la puissance de calcul grandissante des ordinateurs permit progressivement aux développeurs de programmer des logiciels de plus en plus volumineux, en nombre d'instructions. Néanmoins, les méthodes de programmation utilisées restaient artisanales : quand on avait envie de développer un programme, il suffisait de se mettre devant sa machine et de taper des instructions, comme Mozart récitant sa partition. Cette façon de créer un programme sans le concevoir au préalable, sans réfléchir à sa structure et à ses algorithmes, conduisit très rapidement à des programmes qui ne fonctionnaient pas. Soit parce qu'ils contenaient trop de *bugs*[1], soit parce que les développeurs avaient mal compris ce qu'ils avaient à faire ce qui provoque généralement le mécontentement de ceux qui leur ont demandé d'écrire le programme ! Devant une situation de crise, des chercheurs ont commencé à réfléchir à la façon d'écrire "un bon programme", c'est-à-dire un programme qui ne contienne pas de bug et qui fasse exactement ce qu'on attend de lui. Ce que ces chercheurs ont commencé à définir s'appelle le *génie logiciel*. Ce domaine véhicule un paradoxe fondamental : pour bien guider le développeur dans l'écriture de ses programmes il faudrait lui fournir des règles, des indications, très précises sur le *comment faire*, or chaque développement de logiciel est particulier ce

1. Un *bug* peut aussi bien désigner une faute du programme lors de son exécution ou une erreur d'écriture qui fait qu'il ne fait pas exactement ce qu'il devrait.

qui rend ce travail d'aide délicat. On comprend aisément que développer un logiciel de comptabilité pour une PME n'est pas le même exercice que développer un logiciel de pilotage d'une centrale nucléaire ! Ainsi, bien souvent en génie logiciel nous avons tendance à rester au niveau des principes généraux. Libre à chaque développeur de se les approprier et de les mettre en pratique dans son travail. Cet ouvrage présente de manière concrète et illustrée quelques-uns des principes généraux appliqués au langage C++, ce qui vous guidera dans l'écriture de vos programmes.

> Pour résumer on peut dire que le génie logiciel est l'ensemble des règles qu'il faut respecter pour avoir une chance de développer un programme sans trop de bug et que l'on pourra facilement faire évoluer ultérieurement.

1.2 LE CYCLE DE VIE D'UN LOGICIEL

Le cycle de vie d'un logiciel, c'est l'ensemble des étapes qu'un programmeur doit suivre pour créer un logiciel. Il existe de nombreux cycles de vie, presque autant que de programmeurs ! Néanmoins, quelques-uns font référence en la matière et notamment le cycle en V dont le principal avantage est de présenter simplement la démarche de développement. Rassurez-vous, vous ne trouverez pas ici un cours complet sur le cycle en V mais simplement ses grandes caractéristiques (figure 1.1).

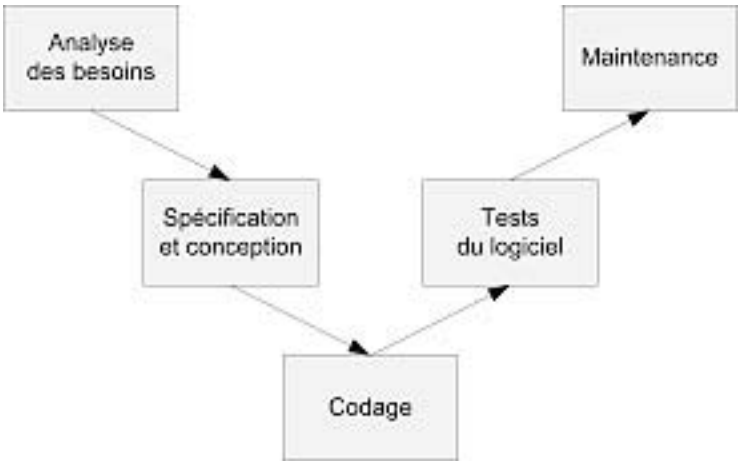


FIG. 1.1: Développer un logiciel en suivant un cycle en V

La première étape à suivre lorsqu'on souhaite écrire un programme, ou plus généralement développer un logiciel, consiste à se poser clairement la question suivante : *Que doit faire mon programme ?* Il est donc nécessaire de formaliser à son besoin, d'écrire un cahier des charges qui énonce ce que va pouvoir faire l'utilisateur du programme, ce que pourra être l'interface graphique, s'il existe des contraintes de sécurité ou de fiabilité... Ce travail est essentiellement un travail de réflexion et d'analyse qui évite bien des déconvenues en pratique lorsqu'on a fini de programmer et de tester son logiciel. Imaginez votre joie si vous passez des heures à programmer quelque chose qui au final ne fait pas exactement ce qu'on vous avait demandé ! Au cours de cette phase, vous allez passer votre temps à écrire des documents de synthèse qui seront la bible du développement par la suite.

Une fois ce travail réalisé, commence la phase de spécification et conception de votre programme, c'est-à-dire le moment où vous allez écrire sur le papier sa structure (Quelles sont les classes à écrire ? Quels sont les méthodes et les attributs ?) et le détail de son fonctionnement (Quels sont les algorithmes des méthodes ?). Ce travail est un travail de spécification et de conception qui vous permet au bout du compte d'avoir un logiciel "sur le papier". Il est généralement réalisé en utilisant des méthodes orientées objet comme UML par exemple. Ce logiciel version papier est ensuite traduit en langage C++ (puisque c'est le langage que nous étudions ici) pour donner naissance, dans la phase de codage, à la première mouture du logiciel. La partie ascendante du cycle en V regroupe les phases de tests et de maintenance. Dans la première, vous allez tester votre logiciel à l'aide de méthodes scientifiques bien définies comme les tests fonctionnels, les tests structurels... La phase de maintenance consiste à faire évoluer votre programme, si nécessaire, une fois que celui-ci est terminé et mis en *production*. Ces évolutions peuvent être de la correction de bugs, de l'amélioration de performances, de l'ajout de nouvelles parties de code...

Cette trame très générale du cycle de vie doit vous permettre de bien comprendre quelle doit être votre démarche lorsque vous développez un logiciel, qu'il soit constitué de 10 ou 50 000 lignes de code. *Il est impératif, avant de se mettre à saisir du code en langage C++ de bien avoir réfléchi, sur le papier, au préalable à la façon dont vous allez vous y prendre*. Même un bout de classe, un morceau de programme écrit sur le coin d'une nappe peuvent être bénéfiques. En tout cas, parfois bien mieux que de créer votre programme en le saisissant directement sous votre environnement de développement : êtes-vous vraiment convaincu que vous allez réussir une bonne recette de gâteau au chocolat en écrivant la recette en même temps que vous le faites pour la première fois ? Vraisemblablement ce gâteau sera de nature expérimentale. Et bien pour la création d'un programme, c'est la même chose : mieux vaut avoir écrit avant tout la recette sur papier avant de se lancer dans l'écriture du programme. L'objectif de cet ouvrage n'est pas de vous faire un cours complet sur le génie logiciel (vous trouverez une introduction plus complète dans la référence [**8**] et plus de détails techniques dans [**2**] et [**9**]) mais simplement de vous alerter sur ce qu'il faut faire ou pas.

Voici quelques repères plus concrets (la liste est loin d'être exhaustive) qui doivent vous aider à juger quand vous faites bien ou pas les choses.

- *Vous êtes sous votre environnement de développement et vous écrivez spontanément du code en langage C++*. Attention, danger ! Vous êtes en train de taper du code en même temps que vous le concevez. Donc, vous résolvez des problèmes de conception en temps réel ce qui n'est jamais bon. Et hop, une nouvelle variable locale par ici, une nouvelle fonction par là... Autant que faire se peut, vous devez éviter cette situation : **tout code tapé doit avoir été pensé au préalable**.
- *Vous êtes en train d'écrire une fonction dont le nombre de lignes de code est supérieur à 100 lignes*. Là encore, vous prenez le risque de faire une fonction qui va être difficilement compréhensible à la relecture car trop volumineuse. Vous augmentez également le risque que la fonction soit une boîte noire qui fait tout et n'importe quoi. Et aussi vous augmentez les risques d'introduction de bugs. **Une fonction doit faire un traitement le plus simple possible**. Un traitement complexe est simplement implémenté grâce à une fonction qui en appelle plusieurs autres plus simples. On préfère donc créer plus de fonctions mais dont la taille est maîtrisée.
- *Vous êtes en train d'écrire une fonction qui ne contient aucun commentaire*. Comment voulez-vous que l'on puisse rapidement comprendre votre fonction lorsqu'on la lira ? **Les commentaires sont là pour faciliter la compréhension d'une fonction**. Une fonction bien commentée doit être aussi facile et agréable à lire qu'un bon roman.
- *Vous avez utilisé une méthode comme UML pour spécifier la structure de votre programme*. Bravo ! Vous avez donc bien défini sur papier quelles étaient les classes que vous alliez programmer, quelles étaient les fonctions du programme et les variables à utiliser. Peut-être avez-vous même écrit sur papier les algorithmes des fonctions ?
- *Chaque classe programmée est testée*. Pour chacune de vos classes, dès qu'elles sont créées, vous instanciez plusieurs objets, appelez leurs méthodes et vous vérifiez que tout se passe bien. C'est bien : vous avez de grandes chances de détecter très vite les bugs les plus grossiers et donc de gagner du temps par la suite.

1.3 SPÉCIFICATION ET CONCEPTION D'UN LOGICIEL

Dans la suite de ce chapitre nous allons voir quelques points importants et fondamentaux que nous appliquerons par la suite lors de votre apprentissage du langage C++. Comme vous n'êtes pas encore censés connaître ce langage, les exemples illustratifs seront donnés en langage algorithmique.

1.3.1. Les commentaires

Les commentaires sont essentiels dans un programme : un bon programme est avant tout un programme qui se lit et se comprend aussi facilement qu'un roman. Ainsi, un code bien commenté sera facilement maintenable : on pourra le corriger ou le faire évoluer sans difficulté.

Souvent, lorsqu'on programme, mettre des commentaires est pris comme une punition. On ne sait jamais quoi mettre comme commentaire et puis surtout : à quoi ça sert, puisque ce n'est pas exécuté par l'ordinateur ? Voici deux versions d'un même programme où les commentaires sont les lignes commençant par le symbole "//". Commencez par lire la première version et essayez de la comprendre. Ensuite, lisez la seconde version.

Algorithme : Version 1

```
entier x,i ;
x = 0 ;
// x est initialisé
Pour i = 1 à 10
// On parcourt la liste de 1 à 10
    x = x + i ** 2 ;
    // On ajoute i ** 2 à x
FinPour;
```

Algorithme : Version 2

```
entier x,i ;
// On calcule la somme de nombres au carré
x = 0 ;
// On calcule pour les nombres de 1 à 10
Pour i = 1 à 10
    x = x + i ** 2 ;
// x = 1**2+2**2+...+i**2
FinPour;
//x = somme i**2, i=1..10
```

La lecture de la première version montre que les commentaires ne servent pas à grand-chose puisqu'ils ne constituent qu'une traduction des lignes de code en langage algorithmique. Donc pour comprendre l'algorithme il vous a été nécessaire de comprendre chaque ligne de code. Ce n'est pas la bonne démarche : *pour comprendre du code il ne doit pas être nécessaire dans votre tête de simuler l'exécution de cet algorithme*. La seconde version est déjà plus correcte : les commentaires décrivent l'état des variables (ici la variable x) et de l'algorithme, ce qui permet de comprendre plus rapidement. *En règle générale, un commentaire doit décrire en français l'état des objets (variables, algorithmes) plutôt que traduire des lignes de code*. De même, les commentaires s'écrivent en même temps que l'on écrit le programme, pas après.

1.3.2. Les exceptions

Les exceptions constituent un outil fondamental pour un développement propre. Entrons dans le vif du sujet.

> Une exception est une interruption volontaire du programme, pour éviter de tomber dans un bug, anticipée par le programmeur lors de l'écriture du code.

Il n'est pas toujours simple de comprendre comment marche le mécanisme des exceptions et de comprendre à quoi il sert. Commençons déjà par expliquer sur un exemple son fonctionnement. Supposons que nous ayons écrit deux fonctions `f` et `g` en langage algorithmique comme indiqué dans la figure 1.2.

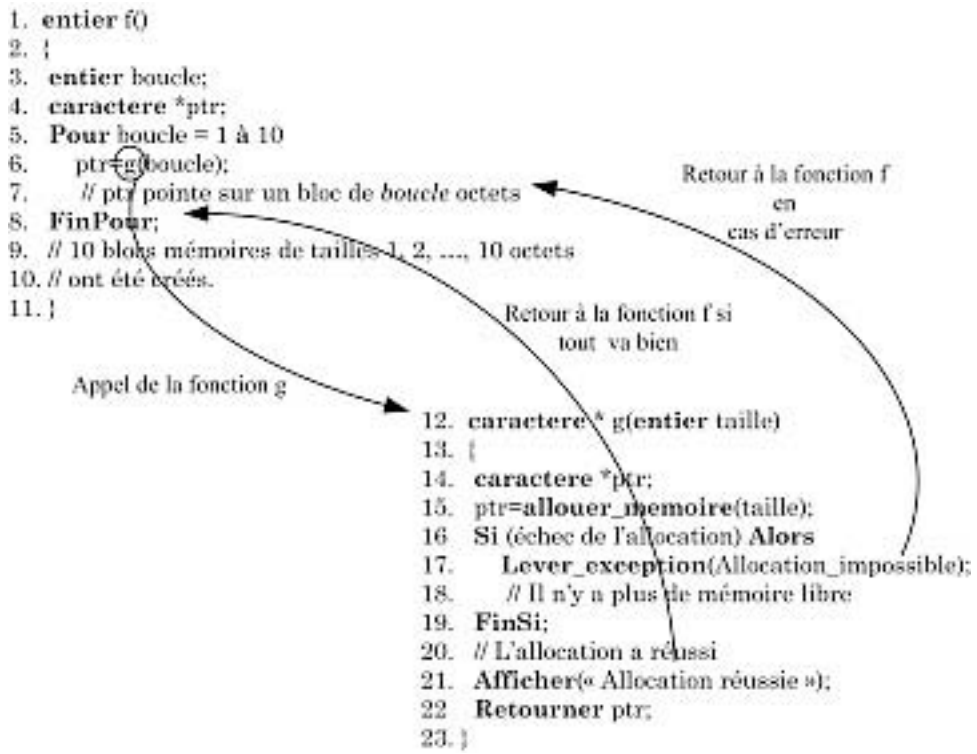

1. entier f()
2. {
3. entier boucle;
4. caractere *ptr;
5. Pour boucle = 1 à 10
6. ptr=g(boucle);
7. // ptr pointe sur un bloc de boucle octets
8. FinPour;
9. // 10 blocs mémoires de tailles 1, 2, ..., 10 octets
10. // ont été créés.
11. }
Retour à la fonction f en cas d'erreur
Retour à la fonction f si tout va bien
Appel de la fonction g
12. caractere * g(entier taille)
13. {
14. caractere *ptr;
15. ptr=allouer_memoire(taille);
16. Si (échec de l'allocation) Alors
17. Lever_exception(Allocation_impossible);
18. // Il n'y a plus de mémoire libre
19. FinSi;
20. // L'allocation a réussi
21. Afficher(« Allocation réussie »);
22 Retourner ptr;
23. }

FIG. 1.2: Un exemple simple d'utilisation des exceptions

La fonction `f` appelle, à la ligne 6, la fonction `g` pour réaliser l'allocation d'un bloc mémoire de *boucle* octets. Il s'avère que le concepteur de la fonction `g` a prévu qu'il pouvait se produire ce que l'on appelle *une situation anormale mais prévisible de fonctionnement* à la ligne 15 de la fonction `g` : en effet, il se peut très bien que l'allocation demandée ne puisse pas être réalisée (car, par exemple, il n'y a plus assez de mémoire disponible). Le concepteur de la fonction `g` prévoit donc deux scenarii possibles : soit la fonction `g` réussit à faire l'allocation et elle doit retourner l'adresse du pointeur alloué, soit l'allocation est impossible et il faut signaler à la fonction f cet échec. Ce signalement est réalisé en levant une exception, en l'occurence l'exception "Allocation_impossible" (lignes 16 à 19). Dans ce cas, la ligne 18 provoque l'arrêt de l'exécution de la fonction `g` (les lignes 20 à 23 ne sont pas exécutées) et le retour à la ligne 7 dans la fonction `f` avec l'exception "Allocation_impossible". À ce moment-là,

soit la fonction f sait comment régler ce problème (et elle continuera de s'exécuter) soit elle ne sait pas et se fera interrompre à son tour.

Dans l'exemple de la figure 1.2, aucune ligne de code spécifique n'a été ajoutée à la fonction `f` après l'appel à `g`, ce qui veut dire que la fonction `f` se fera interrompre si `g` lève l'exception. Donc on retourne à la fonction qui a appelé `f` en lui soumettant l'exception : soit elle sait la contourner et elle continuera de s'exécuter, soit elle ne sait pas auquel cas elle sera interrompue à son tour. On remonte ainsi éventuellement de proche en proche jusqu'à la fonction principale du programme. Si cette fonction ne sait pas gérer l'exception, le programme est interrompu et un message d'erreur spécifique apparaît à l'écran. Au moins, le programme se sera terminé proprement sans bugger. La question qui reste est comment gérer une exception qui vient d'être levée ?

```
1.  entier f()
2.  {
3.   entier boucle;
4.   caractere *ptr;
5.   Pour boucle = 1 à 10
6.      ptr=g(boucle);
7.      EnCasException(Allocation_impossible)
8.           Afficher(« Il n'est pas possible d'allouer de la mémoire »);
9.           Retourner;
10.     FinEnCasException;
11.     // ptr pointe sur un bloc de boucle octets
12.  FinPour;
13.  // 10 blocs mémoires de tailles 1, 2, ..., 10 octets
14.  // ont été créés.
15. }
```

FIG. 1.3: La fonction f gère maintenant l'exception "Allocation_impossible"

Et bien, regardez dans la figure 1.3 le nouveau code de la fonction `f`. Les lignes 7 à 10 ont été modifiées pour inclure du code qui permet "d'attraper" l'exception levée par la fonction `g` et de la traiter, bien que le traitement ici se résume à l'affichage d'un message sur l'écran et à la sortie de la fonction `f`. Au moins, la fonction qui a appelé la fonction `f` pourra continuer de s'exécuter.

Le fonctionnement des exceptions n'a donc plus aucun secret pour vous. Ce qu'il vous reste à apprendre ce sont les instructions du langages C++ pour les mots clefs **LeverException** et **EnCasException**. Nous verrons ça au chapitre 6. En résumé, la mise en place du mécanisme des exceptions implique :

– Pour le concepteur d'une fonction (par exemple, la fonction g) :

  - D'anticiper les situations anormales mais prévisibles de fonctionnement (par exemple, plus assez de mémoire disponible, plus de place sur le disque...).
  - Pour chacune de ces situations, d'ajouter le code de création et de levée d'une exception.
  - D'écrire dans l'en-tête de la fonction la liste des exceptions qui peuvent être levées.
- Pour l'utilisateur d'une fonction (par exemple, la fonction f), et pour chacun des appels de fonction levant une exception :
  - De bien lire l'en-tête de ces fonctions pour savoir si des exceptions peuvent être levées.
  - D'ajouter le code de gestion des exceptions correspondantes.

Le mécanisme des exceptions est primordial dans le cadre d'une démarche de génie logiciel. Il a de multiples avantages :

(1) Il permet d'éviter les arrêts aléatoires de votre programme. Si dans l'exemple précédent, il n'y avait eu aucune exception de levée il y aurait certainement eu des arrêts de votre programme lors de l'utilisation des pointeurs soit disant alloués... mais pas forcément lors de la première utilisation ! Cela aurait entraîné un débugage long et fastidieux.

(2) Il permet de mieux valider et corriger votre programme. Parfois, une erreur de conception peut conduire à la levée d'exceptions qui n'ont pas lieu d'être. Cela vous permet donc souvent, en pratique, de corriger plus vite vos erreurs.

(3) L'utilisation des exceptions facilite la lecture du code des fonctions puisqu'elle met clairement en évidence la gestion des cas particuliers et des situations d'erreur.

Il est vraiment très important d'utiliser les exceptions, même si cela nécessite de votre part une réflexion au préalable sur les situations anormales de fonctionnement pouvant survenir. Ainsi lorsque vous concevez une fonction vous devez systématiquement vous poser la question suivante : **quelles sont les instructions qui peuvent ne pas s'exécuter correctement et provoquer un "plantage" de ma fonction ?** Nous verrons dans la section 1.3.3. comment rédiger l'en-tête d'une fonction et faire apparaître l'existence d'exceptions. De même, nous verrons dans le chapitre 6 comment sont gérées les exceptions en langage C++.

1.3.3. La spécification logique d'une fonction

Lorsque vous écrivez une fonction, que ce soit en langage C ou C++, il est nécessaire d'écrire sa "spécification logique" qui peut être vue comme sa notice d'utilisation. À la lecture de cette spécification, un programmeur doit être capable de :

(1) savoir ce que fait votre fonction,

(2) passer les bons paramètres,

(3) récupérer les valeurs de retour,

(4) savoir si des exceptions peuvent être levées par la fonction.

Il est très facile de savoir si une spécification logique est bien écrite ou pas : si vous devez aller lire le code de la fonction pour chercher une information sur son utilisation, alors c'est que sa spécification n'était pas bien rédigée.

Une spécification logique s'écrit, sous forme de commentaires, selon le formalisme suivant.

E	: Liste des variables d'entrée de la fonction.
nécessite	: Préconditions à respecter sur les variables d'entrée.
S	: Liste des variables de sortie de la fonction.
entraîne	: Postconditions ou ce que fait la fonction.

À titre d'exemple, voici la spécification logique d'une fonction qui calcule le factoriel d'une valeur n donnée en paramètre[2].

fonction entier factoriel(**entier** n)	
E	: n, nombre dont on va calculer le factoriel.
nécessite	: $n \leqslant 12$, car au-delà on ne peut pas calculer le factoriel.
S	: factoriel = la valeur de $n!$.
entraîne	: la valeur du factoriel n est calculée par la fonction.

Il est intéressant de remarquer qu'il y a ici une précondition sur le paramètre n. En effet, la valeur du factoriel calculé doit être stockée dans une variable dont le type est forcément de capacité limitée. Soyons plus précis : la plus grande valeur que peut stocker un entier[3], non signé, est $2^{32} = 4294967296$ ce qui signifie qu'en stockant la valeur du factoriel dans un entier on peut calculer $12! = 479001600$ mais pas $13! = 6227020800$. Vous comprenez donc pourquoi nous sommes obligés de limiter dans une précondition la valeur du paramètre n. Notez bien que **une précondition précise les conditions dans lesquelles la fonction va correctement se dérouler**. Dans l'exemple, l'utilisateur peut très bien appeler la fonction `factoriel` en passant la valeur 25 en paramètre. Néanmoins, à la lecture de la spécification logique il va s'attendre à ce que la fonction ne fonctionne pas correctement : soit elle va s'arrêter sur un message d'erreur, soit le résultat retourné sera incohérent. Normal, la fonction n'était pas prévue pour !

Comme vous commencez à le deviner, les préconditions et postconditions sont des éléments primordiaux, non seulement dans l'écriture des spécifications logiques,

2. Rappelons que le factoriel de n s'écrit : $n! = 1 \times 2 \times 3 \times 4 \times ... \times n$.

3. Un entier est codé sur 4 octets sur la plupart des ordinateurs récents.

mais également dans la conception des fonctions. En effet, **le choix des préconditions et postconditions résulte d'un travail de conception** : c'est vous qui décidez selon ce que vous jugez le plus approprié. Il existe ce que l'on appelle la *dualité précondition/postcondition* qui stipule que ce tout ce qui peut être mis en précondition peut également être mis en postcondition sous forme d'exceptions. Pour illustrer cela, revenons à l'exemple de la fonction `factoriel` qui aurait également pu s'écrire de la façon suivante.

fonction entier factoriel(**entier** n)	
E	: n, nombre dont on va calculer le factoriel.
nécessite	: rien.
S	: factoriel = la valeur de $n!$.
entraîne	: (la valeur du factoriel n est calculée par la fonction) **ou** (**Exception** racine_trop_grande : $n \geqslant 13$).

Dans cette version, l'utilisateur peut appeler la fonction en mettant la valeur qu'il veut pour n. Le programmeur de la fonction `factoriel` aura prévu dans le code un traitement par exception dans le cas où $n \geqslant 13$, de la forme :

```
...
Si n ⩾ 13 Alors
   LeverException ("Racine du factoriel trop grande");
FinSi;
...
```

L'avantage de gérer par une exception en postcondition est que votre fonction sera plus "protégée", puisque vous évitez les risques de mauvaise utilisation (ici, l'utilisateur peut passer la valeur 25 à la fonction `factoriel`, celle-ci ne vas pas s'arrêter sur un message d'erreur). L'inconvénient est que cela force le programmeur à alourdir le code de sa fonction, puisqu'il va devoir ajouter des lignes de code pour la protéger.

La question qui se pose sans doute à vous est de savoir quand il faut mettre une précondition et quand il faut mettre une exception en postcondition. Il n'y a malheureusement pas de règles et cela résulte d'un choix de votre part. Disons que si la condition imposée sur le paramètre est "facile à vérifier" par l'utilisateur alors mieux vaut mettre une précondition. Si jamais cette condition est compliquée ou s'il vous apparaît vital pour le bien de votre programme de protéger votre fonction, mettez une exception en postcondition. Pour conclure, dans l'exemple précédent, qu'est-il pour vous plus approprié de choisir ? Et bien, sans doute, de mettre une précondition car aprés tout il est facile pour l'utilisateur de ne pas appeler la fonction `factoriel` avec une valeur $n \geqslant 13$ et quand bien même il le ferait, cela ne ferait pas planter la fonction. Elle retournerait juste un résultat incohérent.

1.3.4. Une première vision des classes

D'un point de vue génie logiciel une classe est composée de deux parties : une interface (contenue dans un fichier .h ou .hpp) et un corps (contenu dans un fichier .cpp). Par exemple, la classe `toto` va être représentée par deux fichiers : le fichier `toto.h` et le fichier `toto.cpp`. Tout le code effectif de la classe est normalement contenu dans le corps tandis que l'interface contient les déclarations des membres de la classe.

Il est évidemment possible que la classe `toto` inclue une autre classe, par exemple la classe `tata`. La syntaxe d'inclusion est la même qu'en langage C, à savoir :

Fichier `toto.h`

```
#include "tata.h"
...
```

Dans ce cas, toute déclaration effectuée dans l'interface de la classe `tata` est accessible dans la classe `toto`. Par exemple, si vous déclarez une variable dans cette interface, alors elle sera manipulable dans la classe `toto`.

Ainsi, vous ne devez mettre dans l'interface d'une classe que le minimum de déclarations nécessaires pour faire fonctionner votre programme. Cela découle du principe qu'en génie logiciel on appelle le principe *d'encapsulation*. D'un point de vue pratique, cela peut être vu comme un principe de précaution : puisque tout ce qui est mis dans une interface est accessible par inclusion (cf. l'exemple des classes `toto` et `tata`), je dois protéger un maximum mes données et mes traitements en les déclarant le plus souvent possible dans le corps de mes classes ; pour éviter qu'elles soient manipulées par les classes qui incluent les miennes. Bien entendu, vous veillerez à mettre dans l'interface de vos classes les déclarations des variables, types et fonctions dont on a besoin dans les classes qui incluent... mais pas plus !

1.4 QUELQUES RÈGLES DE BONNE PROGRAMMATION

Nous allons voir ici quelques règles générales de génie logiciel qui vont vous guider dans l'écriture de programmes en langage C++. L'objectif est donc de vous les présenter, de vous les expliquer, sachant qu'elles seront illustrées au cours des chapitres qui suivent. Nous distinguons les règles liées à la spécification du programme (les règles qui disent quelles classes concevoir) et les règles liées à la conception du programme (les règles qui disent comment écrire les classes). Ces règles ne sont pas toujours simples à comprendre et surtout à assimiler. Pourtant, c'est en les assimilant et en les acceptant que vous vous forgerez une démarche de programmation fiable.

1.4.1. Règles liées à la spécification du programme

a) Anticiper les évolutions de votre programme

Les règles de *continuité modulaire* [**4**] et *d'encapsulation des modifications* [**6**] énoncent que le découpage en classes que vous allez faire doit anticiper les évolutions futures que votre programme va subir.

> Lorsque vous découpez votre programme en classes vous devez essayer de prévoir les évolutions qu'il subira (Quelles fonctions allez vous ajouter ? Quelles fonctions allez vous modifier ? ...). Ces fonctions devront être regroupées le plus possible au sein de mêmes classes. Les parties susceptibles d'évoluer doivent être mises impérativement dans le corps des classes, les interfaces devant changer le moins possible d'une version du logiciel à une autre.

Ces règles sont très générales et il est parfois, en pratique, difficile de bien comprendre leur sens.
Voici un exemple illustratif : supposons que la première version de votre programme doive imprimer du texte sur une imprimante. Vous prévoyez que la seconde version du logiciel pourra imprimer également du graphique. Si vous appliquez les règles de continuité modulaire et d'encapsulation des modifications lors de l'écriture de la première version du logiciel, vous allez alors créer une classe `imprimante` qui va regrouper toutes les fonctions qui permettent d'imprimer (figure 1.4). Dans cet exemple, on présente la structure d'une classe `A` utilisatrice de la classe `imprimante`.

L'application des règles de continuité et d'encapsulation nous a fait mettre dans le corps de la classe `imprimante` le code des fonctions qui gèrent directement l'impression. Ainsi, par exemple, la fonction `Af` ne fait qu'appeler des fonctions interfaces de la classe `imprimante`. Cela implique que dans une version ultérieure de votre logiciel vous pourrez :

– Changer le code des fonctions de la classe `imprimante` sans changer le code de la méthode `Af` (à condition bien sûr que ces fonctions fassent, au moins, la même chose dans la nouvelle version).
– Ajouter des fonctions dans la classe `imprimante`, sans provoquer de changements dans les classes utilisatrices comme la classe `A`.

On constate donc que l'encapsulation des modifications tend à minimiser les conséquences de changements dans les modules susceptibles d'évoluer. Imaginez, ce qui se serait passé si vous n'aviez pas fait la classe `imprimante` et si c'est directement la fonction `Af` qui avait contenu le code d'accès à l'imprimante (ce même code contenu dans les fonctions `Initialiser_imprimante`,

`Imprimer_caractere...`)! Toute modification de la façon d'imprimer un caractère, par exemple, aurait conduit à modifier toutes les fonctions qui, comme `Af`, impriment des caractères.

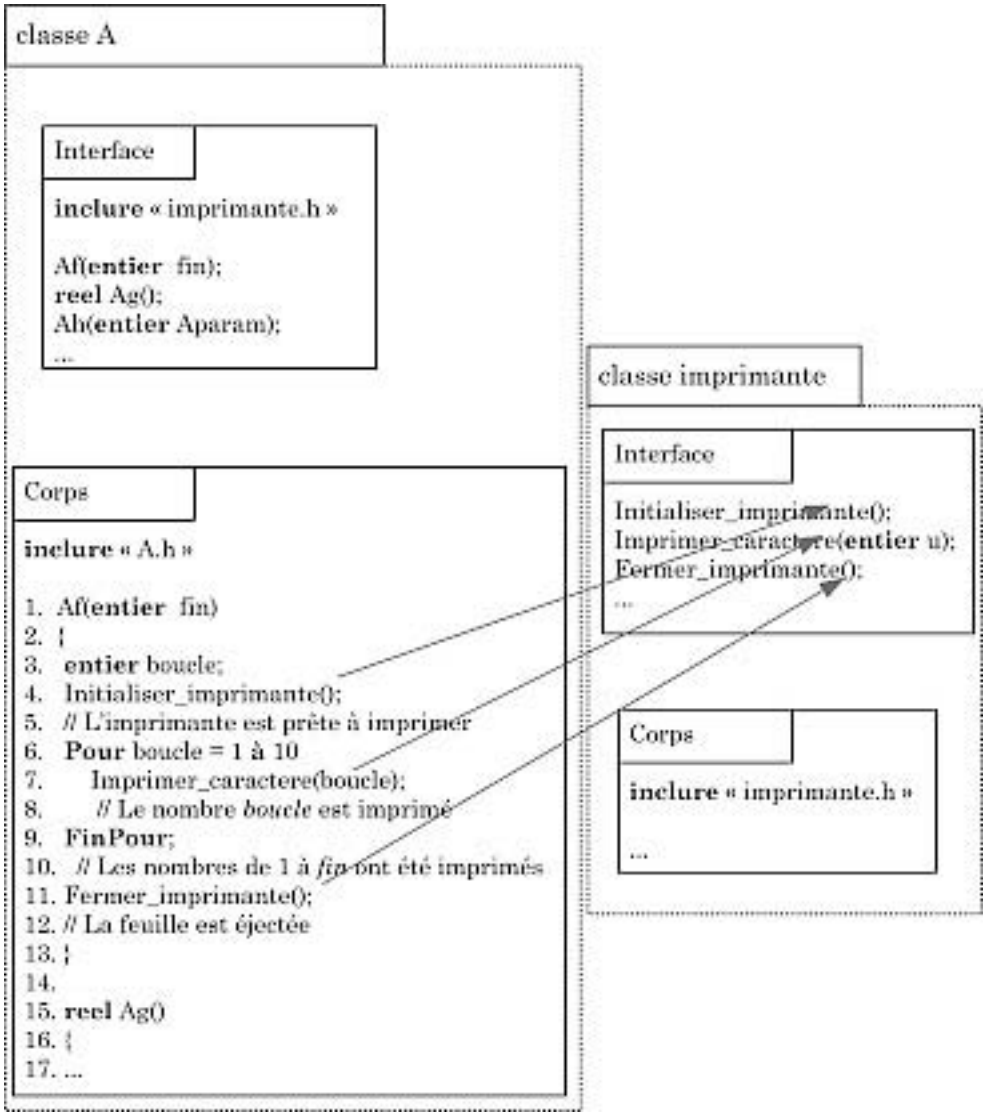


FIG. 1.4: Conséquence des règles de continuité et d'encapsulation

b) Bien séparer les difficultés

La règle de séparation des difficultés [**1**] est une règle qui donne des indications sur les classes à concevoir, ou du moins qui doit vous guider dans le choix de vos classes.

> Lorsque vous découpez votre programme en classes vous devez essayer de prévoir des classes simples. Évitez des classes qui font trop de choses, qui proposent trop de fonctions à l'utilisateur. Une classe doit être vue comme une petite brique de base d'un mur plus grand, votre programme.

Cette règle tend à augmenter le nombre de classes dans un programme, mais cela est plutôt positif. La meilleure analogie que l'on puisse faire est celle d'une fourmilière. Un programme est une fourmilière, chaque classe étant une fourmi : vous n'avez pas de fourmi qui soit en même temps guerrière, ouvrière et reine ! À chacune son rôle. Et bien avec les classes, c'est la même chose. Partez du principe que de base vous avez tendance à complexifier les choses et que les classes que vous projetez de faire doivent être séparées en sous-classes plus simples. Vous ne serez pas loin de la vérité !

c) Limiter les échanges de données entre classes

La règle d'encapsulation des données [**5**] est une règle qui préconise de limiter les échanges de données entre classes. On entend par "échange de données" les paramètres passés d'une fonction d'une classe à une fonction d'une autre classe. Cela peut également être une fonction qui modifie les attributs d'un objet avant d'appeler une fonction sur cet objet. Vous l'aurez compris, cette notion est suffisamment générale pour avoir de nombreuses applications.

> Lorsque vous découpez votre programme en classes vous devez systématiquement privilégier un découpage qui réduit le plus possible les échanges de données entre classes.

Prenons un exemple. Supposons que vous deviez écrire un programme dans lequel il y a une classe `calculs` qui permet de faire des calculs simples sur une liste d'entiers : recherche du minimum de la liste, du maximum, calcul de la somme des éléments... Supposons que vous ayez fait le découpage de cette classe comme indiqué dans la figure 1.5.

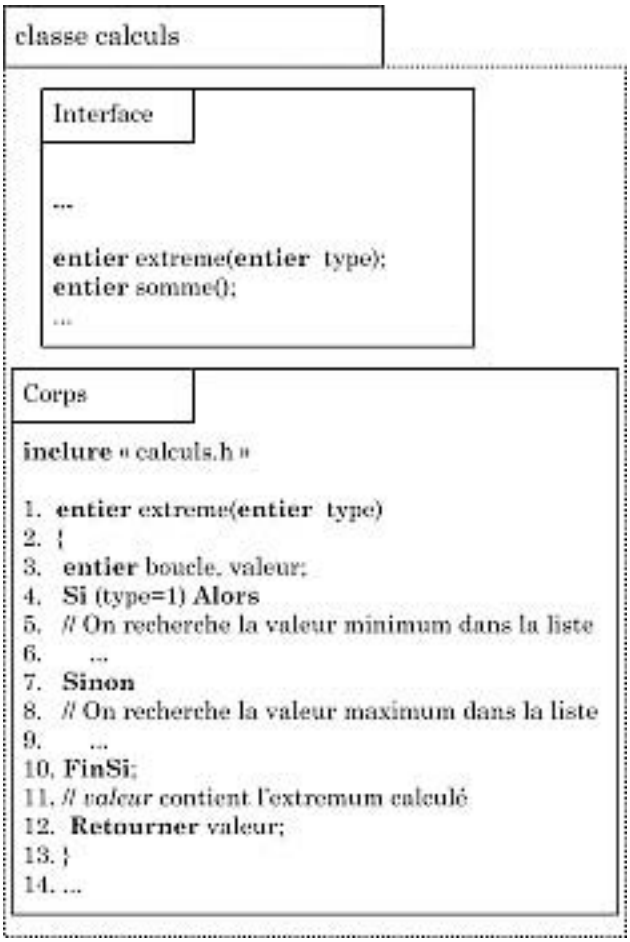


FIG. 1.5: Une première version de la classe `calculs`

Ce découpage ne favorise pas la réduction des échanges de données avec la classe `calculs` car toute fonction qui utilise la fonction `extreme` devra passer un paramètre pour spécifier si on recherche la valeur minimum ou maximum dans la liste gérée dans la classe.

Par ailleurs, on s'aperçoit que la présence de ce paramètre complique l'utilisation de la classe. Il aurait été bien plus simple de créer deux fonctions dans la classe : une fonction `min` qui calcule la valeur minimum et une fonction `max` qui calcule la valeur maximum. La règle d'encapsulation des données va donc nous conduire à réaliser une autre classe `calculs` comme indiquée dans la figure 1.6.

Cet exemple met en évidence le fait que, dans la première version de la classe `calculs` le paramètre `type` de la fonction `extreme` est un paramètre de type

option, c'est-à-dire qu'il ne fait que conditionner le déroulement de la fonction (il n'intervient que dans le **Si**...**Alors**...**Sinon**, lignes 4 à 10 dans la figure 1.5). **Il faut éliminer le plus possible ces paramètres, ce qui tend à multiplier le nombre de fonctions au sein de vos classes**.

Dans notre exemple, la suppression de ce paramètre a conduit à la création de deux fonctions `min` et `max` en remplacement de la fonction `extreme`. Nous avons réduit les flots de données à destination de la classe `calculs`. Cet exemple, va également dans le sens de la règle de *compatibilité ascendante* [**7**] qui préconise de faire des fonctions aux interfaces minimalistes (le moins de paramètres possibles, quitte à multiplier les fonctions) et de prévoir pour chaque attribut de vos classes des fonctions de consultation et de modification (ce qu'on appelle des accesseurs).

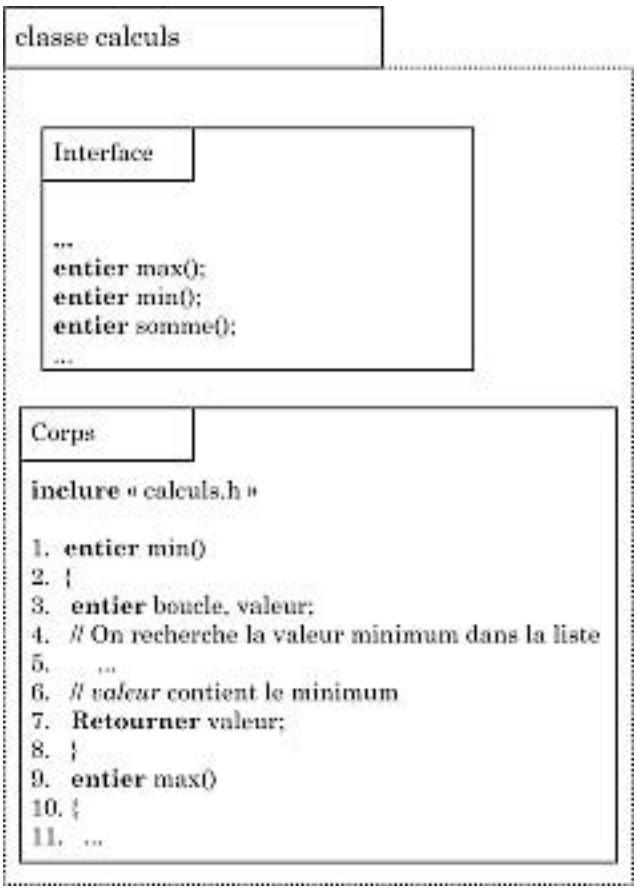


FIG. 1.6: Une seconde version de la classe `calculs`

d) Favoriser la réutilisation des classes

Souvent écrire un programme se fait en partant de rien. Du moins, lorsque vous n'avez aucune démarche de développement structurée. En réalité, beaucoup d'entreprises ayant un minimum de règles de développement cherchent à rentabiliser les développements antérieurs en appliquant notamment la règle de *réutilisation des composants* [**7**].

> Lorsque vous créez vos classes, essayez d'identifier les classes qui peuvent être utilisées dans d'autres programmes. On parle de *classes d'intérêt général*. Pour chacune de ces classes regardez si vous ne les avez pas déjà développées dans un programme antérieur (quitte à les modifier légèrement). Pour les classes d'intérêt général dont ce n'est pas le cas, vous devez prévoir qu'elles seront réutilisées plus tard (vous devez *élargir leur spécification* !).

Cette règle énonce deux choses que nous allons illustrer sur un exemple. Supposons que vous souhaitiez écrire un programme qui va avoir besoin à plusieurs reprises de gérer des listes d'éléments (des nombres entiers, des nombres réels...). En y réfléchissant un peu vous vous faites la réflexion que vous tenez là une classe d'intérêt général : la classe `liste_generique`. En effet, avec une telle classe en main vous allez pouvoir créer dans votre programme plusieurs listes que vous remplirez avec les valeurs que vous voulez. Et puis, il est naturel de penser que votre programme n'est pas le seul qui va manipuler des listes d'éléments. La règle de *réutilisation* vous dit alors de regarder dans les programmes que vous avez déjà écrit en langage C++ pour voir si vous n'avez pas déjà codé une telle classe ou une classe suffisamment proche pour être rapidement adaptable à votre programme. Les avantages de la réutilisation de classes sont nombreux :

(1) Vous réutilisez une classe qui a déjà fait ses preuves. Elle a été testée et validée. Elle résulte éventuellement de l'intervention de plusieurs programmeurs et elle peut être optimisée.

(2) Vous vous épargnez une charge de travail importante puisque vous n'avez pas à refaire le travail de spécification, conception, codage et test.

(3) Vous gagnez du temps de développement et augmentez la qualité de votre programme.

Éventuellement, la classe que vous allez réutiliser va nécessiter quelques ajustements : ajout d'attributs (peut souhaitable quand même), ajout de fonctions ou extension de fonctions existantes. Le plus important est de ne pas modifier les interfaces (nom et liste des paramètres) des fonctions existantes dans la classe ni de modifier les traitements réalisés par ces fonctions (vous y perdriez en compatibilité avec la version antérieure de votre classe).

Si dans aucun de vos précédents programmes la classe `liste_generique` n'a été codée vous devez prévoir que cette classe a de fortes chances d'être réutilisée ultérieurement. Votre travail de spécification et de conception doit être adapté en conséquence, notamment en :

(1) Prévoyant des commentaires clairs dans l'interface et le corps de la classe.

(2) Mettant des accesseurs en lecture et en modification pour chacun des attributs de la classe.[4] Nous reverrons ça ultérieurement dans cet ouvrage.

(3) Vous devez prévoir d'élargir la spécification de vos fonctions. Par exemple, supposons que pour le développement en cours de votre classe `liste_generique` vous ayez besoin d'une fonction **entier** `lire_element()` qui retourne systématiquement le dernier élément de la liste. L'élargissement des spécifications de cette fonction va vous conduire à créer une fonction `lire_element` qui renvoie non pas le dernier élément de la liste mais un élément dont la position est donnée en paramètre. Ainsi, la fonction devient **entier** `lire_element`(**entier** pos) et retourne donc l'élément situé à la position *pos* dans la liste. Comme ça, votre classe `liste_generique` a plus de chance d'être réutilisée par la suite puisqu'elle est un peu plus générale que ce dont vous avez besoin pour le développement en cours.

Pensez réutilisation de classes et commencez dès maintenant à vous constituer une base de données de classes, vous apprécierez le gain ultérieur !

1.4.2. Règles liées à la conception du programme

a) Protéger ses classes

Nous avons vu dans la section 1.3.2. le mécanisme des exceptions qui permet de protéger votre programme en anticipant les situations d'arrêt brutal de votre programme suite à des erreurs. La règle de *protection modulaire* [**4**] préconise non seulement l'utilisation systématique de ce mécanisme mais présente également comment les utiliser **proprement**.

> Toute classe est responsable des erreurs survenant à l'occasion du déroulement d'une de ses fonctions membres. Une spécification d'exception, figurant dans l'interface de la fonction, définit les anomalies dont elle assure l'identification ainsi que la méthode utilisée pour en informer son client (la fonction qui l'utilise).

Cette règle énonce que si une fonction peut lever une ou plusieurs exceptions lors de son exécution, la liste des situations d'exception doit figurer dans sa spécification

4. Un accesseur est une fonction qui permet à une fonction qui manipule des objets de la classe, d'accéder soit en modification soit en lecture à un des attributs de la classe.

logique (cf. section 1.3.3. et l'exemple de la fonction `factoriel`). En reprenant l'exemple des fonctions `f` et `g` de la section 1.3.2., la règle de protection modulaire indique de faire figurer dans la spécification logique de la fonction `g` :

```
...
entraîne    : ...
                (Exception Allocation_impossible : plus assez de mémoire libre)
...
```

De même il faudra faire apparaître cette spécification logique dans l'interface de la classe qui contient la fonction `g` et prévoir la définition de l'exception *Allocation_impossible* (par exemple, en associant une valeur numérique à cette exception, cf. annexe E). L'objectif est qu'à la simple lecture de l'interface de la classe contenant `g` (n'oubliez pas qu'une interface de classe doit se lire aussi facilement qu'un bon roman !), l'utilisateur sache quelle exception peut être levée, par quelle fonction et quelle est la valeur de chaque exception. Nous verrons dans le chapitre 6 qui porte sur les exceptions en langage C++ des exemples d'applications de cette règle. Retenez-en le principe c'est déjà bien.

b) Mettre ses fonctions à angle droit

La règle *d'orthogonalité* [**7**] est plus compliquée qu'il n'y paraît à comprendre, notamment parce qu'il est difficile de voir en pratique ce qu'elle implique. Commençons tout d'abord par l'énoncer telle qu'elle est.

> Chaque fonction doit réaliser un traitement simple. Les différentes fonctions doivent être les plus indépendantes possibles tant dans leur fonctionnement que dans l'ordre dans lequel elles s'appellent. Les cas de dépendance sont explicitement précisés dans l'interface de leur classe d'appartenance.

Une fonction doit être conçue comme une petite brique de base d'un ensemble plus grand. Une brique fait-elle un mur ? Non, pour faire un mur il faut de nombreuses briques, compactes et bien solides. Et bien pour un programme c'est pareil. Gardez bien à l'esprit qu'une fonction doit faire quelque chose de simple et qu'elle ne doit pas contenir "trop" de lignes de code (pas plus de 100 lignes). Mieux vaut parfois faire plusieurs petites fonctions qu'une seule grosse fonction. Ainsi, vous diminuez les risques de bugs, l'utilisateur de votre fonction sait plus facilement ce qu'elle fait et comment s'en servir, ce qui diminue également les risques de mauvaise utilisation de sa part.

Cette règle précise également que si une fonction dépend de l'exécution d'une autre fonction, vous devez également le mentionner dans l'interface de la classe d'appartenance. L'objectif est d'informer l'utilisateur, bien que cela ne soit pas fondamental.

c) Normaliser l'écriture des classes et fonctions

Il s'agit d'un point plus important qu'il n'y paraît. Vous devez absolument structurer la façon dont vous écrivez votre code et ce sur plusieurs points : nommage des variables et types, nommage des fonctions et des classes, en-tête normalisés pour vos fonctions et vos classes... Les normes de rédaction que vous allez utiliser doivent favoriser la mémorisation et la lecture ultérieure de votre code. Les normes de rédaction que nous allons voir dans la suite de cette section ne sont ni exhaustives, ni exclusives. Elles ne sont qu'une proposition et peuvent être améliorées ou modifiées selon votre propre expérience.

Commençons tout d'abord par le *nommage des variables, types, fonctions et classes* et présentons l'ensemble des conventions que nous allons utiliser par la suite.
Nous nommerons tous **les types** en commençant par la lettre `T` en majuscule et suivi du nom en minuscule, par exemple, `Ttableau`, `Tchaine`...
Nous nommerons **les classes** de la même façon en commençant par la lettre `C` en majuscule suivi du nom en minuscule, par exemple, `Cliste_generique`, `Cimprimante`...

Lettre	Type de base
`c`	**char**
`uc`	**unsigned char**
`i`	**int** (ou **long int**)
`si`	**short int**
`usi`	**unsigned short int**
`f`	**float**
`d`	**double**
`ld`	**long double**
`b`	**boole**

TAB. 1.1: Correspondances entre nom de variable et type associé

Concernant les noms de fonctions nous utiliserons les trigrammes pour les fonctions appartenant à des classes. Le trigramme sera généralement constitué par les trois premières lettres du nom de la classe, en excluant la lettre `C` ajoutée comme indiqué ci-dessus. Lorsque le nom de la classe est composé de plusieurs mots vous pouvez faire varier cette règle en prenant des lettres dans plusieurs mots. Par exemple, la fonction `lire_element` de la classe `Cliste_generique` que nous avons déjà vu s'appellera en réalité `LIGlire_element`. Si cette fonction n'appartient pas à une classe (la règle pour les structures est identique au cas des classes) alors on utilisera simplement son nom en le faisant commencer par une majuscule.
Le cas des variables est un peu plus complexe. Chaque nom de variable sera précédé d'une ou plusieurs lettres selon le cas de figure. Si cette variable est un *pointeur* alors

son nom commencera pas la lettre `p` suivi de lettres pour préciser le type sur lequel elle pointe. On utilisera le tableau 1.1 pour les correspondances.

Par exemple, une variable de nom `boucle`, de type **int**, sera nommée `iBoucle`. Une variable de nom `ligne` et de type pointeur sur un **char** sera nommée `pcLigne`.

Si la variable est un attribut d'une classe on utilisera le trigramme associé à la classe (cf. ci-dessus et le nommage des fonctions). Ainsi, si la variable `pcLigne` appartient à la classe `Cliste_generique`, son nom devient `pcLIGligne`.

Attaquons-nous maintenant aux normes de rédaction d'une classe. Vous pouvez trouver dans les annexes E et F deux exemples de classes écrites selon la norme qui sera utilisée tout au long de cet ouvrage. Bien sûr, vous pouvez adapter cette norme en fonction de vos besoins, de votre expérience... il ne s'agit ici que d'une base de départ.

Découvrons cette norme au travers de l'analyse de la classe `Cexception` de l'annexe E et commençons par la description de l'interface. Celle-ci débute par un bandeau (lignes 1 à 21) qui reprend des informations synthétiques sur la classe : un titre, un nom d'auteur ainsi qu'un numéro de version et une date (à répéter autant de fois qu'il y a eu de modifications de la classe), un nom de lecteur et une date de relecture, et puis pour terminer un descriptif de la classe. Il est important de noter que la conception d'une classe s'inscrit dans une démarche auteur/lecteur, c'est-à-dire que l'auteur est celui qui écrit la classe, celle-ci étant relue par un autre programmeur, le lecteur. L'objectif est de détecter très tôt (à la lecture) les erreurs de conception, les bugs... La partie auteur/lecteur est répétée autant de fois qu'il y a eu d'interventions pour modifier ou faire évoluer la classe.

Après la définition du nom de la classe (ligne 26), un commentaire vient décrire ce qu'elle représente (lignes 28 et 29). Ce commentaire est suivi de la définition des attributs de la classe.
Sur les lignes 35 et 36 figurent la définition de l'état initial de chaque attribut de la classe. L'état initial d'un attribut est la valeur qu'il prend lorsqu'un objet de la classe est créé.

À partir de la ligne 38 viennent les déclarations de chacune des fonctions de la classe, également appelées primitives ou méthodes. Chaque déclaration est suivie de la spécification logique de la méthode (cf. section 1.3.3. concernant les spécifications logiques).

Le corps de la classe contient le même bandeau que l'interface. Il est suivi d'une large partie de commentaires (lignes 24 à 34 dans l'exemple de la classe `Cexception`) décrivant les grandes lignes de la structure de la classe. Le champ *Attribut* (ligne 26) contient la description de chacun des attributs déclarés dans l'interface. Le champ *Structure* (lignes 27 et 28) présente le contenu de la classe ainsi que les particularités à connaître concernant son organisation (présence de sous-classes, de classes mères...). Le champ *Méthode* (ligne 29) contient une description des méthodes particulières implémentées dans des méthodes de la classe. Par exemple, si

une fonction repose sur un algorithme clairement identifié dans la littérature ou dans un précédent projet, vous devez le mentionner dans ce champ. Le champ *Modules internes* (lignes 30 à 32) contient la liste des inclusions dont a seulement besoin le corps de la classe. Généralement, on fait au moins figurer l'inclusion de l'interface et les inclusions des fichiers dont on a besoin que dans le corps.

Chapitre **2**

Généralités sur le langage C++

2.1 MOTS-CLEFS, INSTRUCTIONS ET COMMENTAIRES

Le langage C++ comporte un certain nombre de mots-clefs et d'instructions qui ne sont pas redéfinissables. C'est-à-dire que, par exemple, il ne vous est pas possible de créer une variable qui s'appelle `class` puisqu'il s'agit d'un mot-clef du langage. La liste des mots-clefs et des instructions du langage C++ est donnée dans l'annexe A. De même, l'annexe B récapitule l'ensemble des types de données disponibles. Ce langage reprend l'essentiel du langage C en y ajoutant un jeu d'instructions spécifiques aux objets et classes. Néanmoins, il existe tout de même quelques spécificités du langage C++ qui le différencient du langage C. Nous allons voir dans ce chapitre quelles sont ces spécificités.

Tout d'abord concernant les *commentaires*, le langage C++ reprend ceux du langage C, à savoir : une zone de commentaires commence par `/*` et finit par `*/`. Par exemple, dans le code qui suit, les lignes 2 et 3 forment un bloc de commentaires et ne sont donc pas compilées.

```
1 ...
2 /* Nous allons insérer la valeur elem
3  à la position pos dans le tableau pTeLISliste */
4  uiLIStaille++;
5  pTeLISliste=pTetmp;
6  for (iBoucle=uiLIStaille;iBoucle>pos;iBoucle--)
7          pTeLISliste[iBoucle]=pTeLISliste[iBoucle-1];
```

```
8  pTeLISliste[pos]=elem;
9  // L'élément est inséré dans la liste à la position demandée
10 ...
```

Le langage C++ ajoute par contre les **commentaires de fin de ligne** qui commencent par le symbole // et se terminent à la fin de la ligne sur laquelle ils ont commencé. Dans le code ci-dessus, la ligne 9 (et uniquement celle-ci) est un commentaire de fin de ligne. Un tel commentaire peut très bien être placé à la fin d'une ligne contenant du code effectif, comme indiqué dans la portion de code ci-dessous (ligne 2).

```
1  ...
2  uiLIStaille++; // J'incrémente le nombre d'éléments du tableau
3  pTeLISliste=pTetmp;
4  for (iBoucle=uiLIStaille;iBoucle>pos;iBoucle--)
5          pTeLISliste[iBoucle]=pTeLISliste[iBoucle-1];
6  pTeLISliste[pos]=elem;
7  // L'élément est inséré dans la liste à la position demandée
8  ...
```

De même, on peut mélanger les commentaires avec les commentaires de fin de ligne. Dans ce cas, ce sont nécessairement les commentaires du langage C qui l'emportent. Ainsi, il n'est pas gênant qu'à l'intérieur d'un bloc de commentaires compris dans les délimiteurs /* et */ soient placés des commentaires de fin de ligne : ceux-ci ne changeront pas les limites de la zone de commentaires.

```
1  ...
2  /* Voici une zone de commentaires qui inclut
3  // un commentaire de fin de ligne
4  ce qui ne change pas le fait que toutes les lignes ici
5  sont des commentaires */
6  ...
```

2.2 LA GESTION DES VARIABLES EN LANGAGE C++

La gestion des variables en langage C++ est quasiment identique à celle du langage C. Cependant on peut noter quelques changements que nous détaillons dans les sous-sections qui suivent. Ces changements ont été inclus pour apporter plus de souplesse pour les programmeurs.

2.2.1. Déclaration et initialisation des variables

La *déclaration des variables* en langage C++ a été assouplie, dans la mesure où elle peut se faire à l'endroit où l'on a besoin de la variable (et pas seulement au début des blocs comme c'est le cas en langage C). *La portée d'une variable reste limitée au bloc contenant sa déclaration.* On peut aussi déclarer une variable à l'intérieur d'une structure de contrôle (une structure de contrôle est une boucle `for`, un test `if`...). Dans ce cas, et selon la norme ANSI du langage C++, le compilateur considère qu'elle n'a été déclarée que pour le bloc. Sa portée est donc limitée au bloc associé à la structure de contrôle. Examinons la portion de code ci-dessous.

```
1  #include <stdio.h>
2
3  void main()
4  {
5   int iBoucle;
6
7   for (iBoucle=0;iBoucle<10;iBoucle++)
8   {
9    int iBoucle2;
10   iBoucle2=iBoucle+1;
11  }
12  printf("Valeur de iBoucle : %d\n",iBoucle);
13 }
```

La variable `iBoucle2` déclarée à la ligne 9 n'est référençable qu'à l'intérieur du bloc `for`, donc entre les lignes 8 et 11. Si, ligne 12, vous affichiez la valeur de cette variable au lieu de la variable `iBoucle` vous auriez alors un message d'erreur à la compilation : identifiant `iBoucle2` inconnu. De même, si vous remplacez les lignes 5 à 7 par le code suivant :

```
for (int iBoucle=0;iBoucle<10;iBoucle++)
```

vous obtenez un message d'erreur à la compilation de la ligne 12 : identifiant `iBoucle` inconnu. **Il s'agit ici d'une règle uniquement valable dans la norme ANSI du langage C++ et qui n'est pas forcément en vigueur dans tous les compilateurs**, notamment les anciens compilateurs. Pour ceux qui ne respectent pas cette norme, la boucle `for` ci-dessus et les lignes 5 à 7 sont équivalentes puisqu'on considère que la variable `iBoucle` déclarée dans le `for` est accessible dans tout le bloc contenant la boucle `for` (et non plus simplement dans le bloc associé à cette boucle).

D'un point de vue génie logiciel, la déclaration "sauvage" des variables au sein des fonctions est fortement déconseillée, pour des raisons de facilité de lecture du

code. Il est recommandé de toujours déclarer ses variables au début de la fonction de sorte à ce que le lecteur de votre fonction puisse s'y référer rapidement en cas de besoin.

2.2.2. Portée et visibilité des variables

La portée d'une variable est la zone dans laquelle cette variable est connue et est utilisable. En langage C comme en langage C++, elle s'étend depuis l'endroit où elle est déclarée jusqu'à la fin du bloc qui la déclare, y compris les sous-blocs inclus dans ce dernier. Cependant, dans certains cas la variable peut être masquée, par une variable de même nom et appartenant à un sous-bloc inclus. Nous illustrons cela dans l'exemple ci-dessous.

```
1  #include <stdio.h>
2
3  void main()
4  {
5   int iBoucle;
6
7   for (iBoucle=0;iBoucle<5;iBoucle++)
8   {
9    int iBoucle2=iBoucle+1;
10   if (iBoucle2==2)
11          {
12           int iBoucle=3;
13           printf("Valeur de iBoucle : %d\n",iBoucle);
14          }
15   printf("Valeur de iBoucle : %d\n",iBoucle);
16  }
17  printf("Valeur de iBoucle : %d\n",iBoucle);
18 }
```

On remarque que la variable `iBoucle` déclarée dans le bloc compris entre les lignes 5 à 18 a une *portée* égale à ce bloc, c'est-à-dire qu'elle est créée sur la ligne 5 et détruite sur la ligne 18. Elle est donc référençable, a priori, dans les sous-blocs inclus qui sont les blocs associés au `for` de la ligne 7 et au `if` de la ligne 10. Sa *visibilité* est moindre puisque cette variable est masquée par celle de même nom déclarée ligne 12 : ainsi dans le bloc compris entre les lignes 11 et 14, c'est la variable `iBoucle` déclarée ligne 12 qui est référencée et non pas celle de la ligne 5. Cela implique que l'affichage résultant de ce code est le suivant :

```
Valeur de iBoucle : 0
Valeur de iBoucle : 3
Valeur de iBoucle : 1
Valeur de iBoucle : 2
Valeur de iBoucle : 3
Valeur de iBoucle : 4
Valeur de iBoucle : 5
```

Par contre, dès que l'on sort du bloc `if` la variable `iBoucle` "reprend" la valeur qu'elle avait avant d'entrer dans ce bloc (ici, la valeur 1).

Les exemples précédents font ressortir qu'il existe deux catégories de variables. Il y a les variables dites *locales* à un bloc (on parle également de variables automatiques), comme les variables `iBoucle` et `iBoucle2`. Il y a également les variables dites *globales* au programme, c'est-à-dire des variables accessibles dans toutes les fonctions du programme. Il est possible de déclarer qu'*une variable locale soit globale au bloc qui la déclare*. Pour illustrer cela, considérons le code précédent et la variable `iBoucle2` déclarée et initialisée sur la ligne 9. Cette variable est locale au bloc `for` dans lequel elle est déclarée, ce qui veut dire qu'elle est créée par le compilateur sur la ligne 8 et détruite sur la ligne 16. Pour les 5 exécutions du bloc `for` il y aura donc 5 variables `iBoucle2` de créées initialisées avec les valeurs 1, 2, 3, 4 et 5. Il est possible *de déclarer que la variable `iBoucle2` est globale pour le bloc `for`*, c'est-à-dire que pour les 5 exécutions de ce bloc une seule variable `iBoucle2` sera créée. Cette création sera faite par le compilateur lors du premier passage dans le bloc et sera détruite lors du dernier passage dans le bloc. Ainsi, la variable `iBoucle2` dans l'exemple prendra uniquement la valeur 1 puisqu'elle ne sera initialisée qu'une seule fois. La conséquence est que les lignes 10 à 14 ne seront jamais exécutées. Comment déclarer qu'une variable est globale à un bloc ? Il suffit de la déclarer `static` :

`static` type nom_variable ;

Ce qui donne dans notre exemple : "`static int iBoucle2=iBoucle+1;`" sur la ligne 9. Faites la modification dans l'exemple précédent et vous obtiendrez l'affichage suivant :

```
Valeur de iBoucle : 0
Valeur de iBoucle : 1
Valeur de iBoucle : 2
Valeur de iBoucle : 3
Valeur de iBoucle : 4
Valeur de iBoucle : 5
```

Pour terminer sur la notion de variable statique, notez qu'une variable locale à une fonction qui est déclarée statique est accessible et conserve sa valeur pour toutes les exécutions de la fonction : elle peut donc être vue comme une variable globale au programme uniquement référençable dans la fonction qui l'a déclarée.

2.3 NOTION DE RÉFÉRENCE

La notion de référence est propre au langage C++ et n'existe pas en langage C. On distingue deux types de référence : le passage par référence d'arguments à une fonction et la déclaration de variables références.

2.3.1. Passage d'arguments par référence à une fonction

Puisque vous connaissez déjà le langage C, vous devez maîtriser le passage d'arguments par valeur et le passage d'arguments par adresse. Si ce n'est pas le cas, vous pouvez consulter l'annexe C qui explique ces deux types de passage d'arguments. Le passage d'arguments par référence est une troisième façon de faire, qui se situe à mi-chemin entre le passage par valeur et le passage par adresse.

> En résumé, le passage par référence utilise la syntaxe du passage par valeur et met en œuvre le mécanisme du passage par adresse.

Ainsi, à un détail près, passer un argument par référence à une fonction se fait en utilisant la même syntaxe que le passage par valeur. Par contre, le compilateur ne passe pas réellement la valeur de l'argument mais son adresse. Voyons tout d'abord la syntaxe avant de détailler le processus.

```
1  #include <stdio.h>
2
3  void f(int iP1, int * piP2, int & iP3)
4  {
5          // iP1 représente la valeur de l'argument passé
6          // piP2 pointe sur la valeur de l'argument
7          // *piP2 représente la valeur de l'argument
8          // iP3 représente la valeur de l'argument,
9  }
10
11 void main()
12 {
13  int iV1=5,iV2=5,iV3=5;
14  f(iV1,&iV2,iV3);
15    // iV1 est passé par valeur
16    // iV2 est passé par adresse
17    // iV3 est passé par référence
18 }
```

Pour **déclarer qu'un argument est passé par référence** à une fonction, il suffit de placer dans l'interface de la fonction le symbole & devant le nom de l'argument

(cf. ligne 3 et l'argument `iP3`). **L'utilisation d'un argument passé par référence** se fait comme s'il avait été passé par valeur : on manipule directement la valeur. De même, c'est la notation pointée (... `iP3.XXX` si cet argument avait été une structure, par exemple, pour accéder au champ `XXXX`...) qui s'applique pour les objets et structures passés par référence et non la notation flêchée (... donc pas de syntaxe du style `iP3->XXX`...). La ligne 8 du code précédent montre que pour manipuler la valeur de `iP3` dans la fonction `f` on utilise simplement `iP3` et non pas `*iP3`. De même, la ligne 14 montre que passer une variable par référence à une fonction (variable `iV3`) se fait comme pour passer une variable par valeur (variable `iV1`).

Comment est implémenté le passage par référence par le compilateur ? Pour illustrer ce mécanisme, reprenons l'exemple de l'appel à la fonction `f` dans le code précédent et illustré dans la figure 2.1 (attention, la numérotation des lignes a changé dans cette figure). Sur la ligne 10, la variable `iV3` est créée en mémoire et initialisée avec la valeur 5 (action (1)). Ligne 11, cette variable est passée par référence : le compilateur va automatiquement récupérer l'adresse de cette variable et la passer à la fonction (action (2)). Le code de celle-ci manipulera donc directement la variable d'origine `iV3`, comme pour le passage par adresse et contrairement au passage par valeur où une copie de la variable de départ est réalisée. Ainsi la ligne 4 de la figure 2.1 va provoquer le remplacement en mémoire de la valeur 5 par la valeur 10 (action (3)) ce qui implique que sur la ligne 12, après l'exécution de `f`, la variable `iV3` vaudra 10.

Le passage par référence a un certain nombre d'avantages :

(1) Il évite la recopie d'objets comme dans le cas du passage par valeur ce qui provoque un gain de temps lors de l'exécution de votre programme et un gain de mémoire.

(2) Il possède une syntaxe simple qui est celle du passage par valeur.

Toutefois, il y a un inconvénient, essentiellement d'un point de vue *génie logiciel* : **les utilisateurs d'une fonction utilisant le passage par référence doivent se souvenir à chaque appel que l'argument qu'ils passent peut être directement modifié dans cette fonction**. Il s'agit surtout ici d'un problème de mémorisation pour le programmeur qui ne doit pas oublier que de telles modifications peuvent avoir lieu. Il est toutefois possible de spécifier, dans l'interface des fonctions, que les arguments passés par référence ne peuvent pas être modifiés dans la fonction ; il suffit de déclarer ces arguments avec le mot-clef `const` :

type_retour nom_fonction(...,`const` type_argument & nom_argument,...) ;

Ce qui donne dans notre exemple : "`void f(int iP1, int *piP2, const int &iP1)` ;". Par voie de conséquence, le code donné dans la figure 2.1 ne compile plus à cause de la ligne 4 et on obtient le message d'erreur suivant : la valeur de gauche est un objet constant (supposé donc non modifiable).

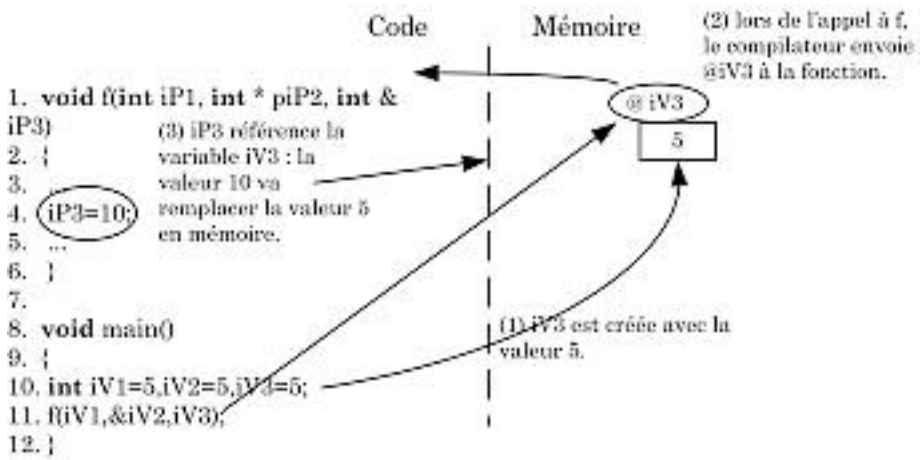


FIG. 2.1: Le mécanisme du passage d'arguments à une fonction par référence

Pour terminer sur le passage d'arguments par référence, voici un petit casse-tête. Supposons que l'appel à la fonction `f` dans l'exemple précédent soit réalisé comme suit dans la première version du code donné dans cette section :

```
f(iV1,&iV2,5);
```

D'après vous, est-ce que cet appel compile ? Si vous reprenez les explications de la figure 2.1, vous en déduisez que la réponse est forcément *non*. Vous aurez le message d'erreur suivant sur la ligne qui appelle `f` : impossible de convertir le troisième argument d'un `const int` vers un `int &`. L'explication est que le compilateur n'est pas capable, à partir de la valeur 5, de retrouver l'adresse : en effet une valeur numérique n'est pas une variable, elle n'est rattachée à aucun emplacement mémoire. Pour éviter ces petits soucis, il suffit de déclarer que l'argument passé par référence (ici `iV3`) est de type constant (mot-clef `const`). L'appel ci-dessus sera alors réalisable... au prix de la création par le compilateur d'une variable temporaire dans laquelle sera mise la valeur 5 et dont l'adresse sera passée à la fonction `f`. Évidemment, cette variable temporaire n'est pas modifiable dans la fonction.

2.3.2. Les variables références

La notion de référence s'applique également sur des variables locales ou globales. Cependant il ne faut absolument pas la confondre avec le passage d'arguments par référence à une fonction. Dans ce cas la variable "référence" pointe nécessairement sur une variable existante. Par ailleurs, on ne peut pas faire pointer une référence déjà définie sur une autre variable. La syntaxe de déclaration d'une référence sur une variable est la suivante :

```
type_reference & nom_reference = nom_variable;
```

Ce qui donne sur un exemple (ligne 3) :

```
1 ...
2 int iV=12;
3 int &iR=iV;
4 ...
```

On notera que le type de la référence doit être le même que le type de la variable référencée car sinon le compilateur met un message d'erreur. Ainsi, il n'est pas possible à la ligne 3 de l'exemple précédent de déclarer `iR` en autre chose qu'un `int`.

2.4 L'EN-TÊTE D'UNE FONCTION

Dans la déclaration de l'en-tête (ou interface) d'une fonction il y a quelques changements entre les langages C et C++.

(1) Si une fonction `f` ne possède pas de paramètre, alors en langage C on doit déclarer :
`type_retour f(void)`
tandis qu'en langage C++ on **peut** déclarer :
`type_retour f()`.

(2) Si une fonction `f` ne retourne aucune valeur, alors en langage C on peut écrire :
`f(liste_de_parametres)`
tandis qu'en langage C++ on **peut seulement** déclarer :
`void f(liste_de_parametres)`.

Note : certains compilateurs C++, en l'absence de type de retour, vont automatiquement supposés que le type est `void` ce qui provoquera juste un message d'avertissement à la compilation. Néanmoins, si vous mettez une instruction de type `return X;` en fin de fonction `f` sans avoir précisé de type de retour, alors `X` doit nécessairement être de type `int` sous peine d'avoir un message d'erreur à la compilation. En effet, si le compilateur doit supposer que votre fonction retourne quelque chose, le type par défaut est toujours `int` sans autre précision de votre part.

2.5 ÉVITER LES PROBLÈMES D'INCLUSIONS MULTIPLES D'INTERFACES

Dans cette section nous nous attardons sur un point qui n'est pas véritablement lié au langage C++ mais qui peut vite devenir problématique lorsqu'on programme en langage C ou C++. En effet, un problème qui revient souvent lorsqu'on écrit des

programmes volumineux est celui de l'inclusion multiple des interfaces des classes. Considérez l'exemple suivant de trois fichiers `.h` : le fichier `A.h` contient la définition d'une structure `Sa` à l'aide du mot-clef `struct`, les fichiers `B.h` et `C.h` incluent le fichier `A.h`.

```
1  Fichier A.h :
2  ————————
3  ...
4  struct SA {
5  ...
6  };
7  ...
```

```
1  Fichier B.h :
2  ————————
3
4  #include "A.h"
5  ...
```

```
1  Fichier C.h :
2  ————————
3
4  #include "A.h"
5  ...
```

```
1  Fichier main.cpp :
2  ——————————
3
4  #include "A.h"
5  #include "B.h"
6  #include "C.h"
7
8  ...
```

À la compilation vous obtiendrez deux messages d'erreur situés sur la ligne 5 du fichier `A.h` : structure `Sa` redéfinie. Cela provient du fait que lors de la compilation des fichiers `A.h`, `B.h` et `C.h` (et par le jeu des inclusions), le compilateur est tombé trois fois sur la définition de la structure (ligne 4 du fichier `A.h`). En effet, puisque le fichier `B.h` inclut le fichier `A.h` le compilateur fait "comme si" la structure `Sa` était définie également dans le fichier `B.h`. Or aux deux derniers passages, la structure était déjà définie ce qui a provoqué l'erreur.

Ces problèmes d'inclusion sont relativement fréquents. Pour les empêcher on utilise des commandes du préprocesseur, c'est-à-dire des instructions qui ne sont pas compilées mais qui guident simplement le compilateur dans son travail. Vous connaissez certaines de ces instructions puisqu'elles commencent toutes par le symbole #, comme par exemple `#include`. Pour éviter ces erreurs d'inclusion, il suffit simplement, dans chaque fichier d'interface `.h` de rajouter les commandes du préprocesseur suivantes :

Fichier `A.h`
#ifndef AH
#define AH 0
... contenu normal de l'interface
#endif

À la première lecture du fichier `A.h` le compilateur définira le symbole `AH` à la valeur 0. À partir de la seconde lecture de ce fichier, lors de la compilation, le symbole `AH` étant défini le compilateur sautera directement à la commance `#endif` ce qui évitera de rencontrer une autre fois les symboles définis dans l'interface normale (dans notre exemple la structure `Sa`). Évidemment le nom `AH` et la valeur 0 n'ont aucune importance... si ce n'est que vous ne devez pas utiliser deux fois le même nom dans des interfaces différentes sous peine d'avoir de drôles de surprises !

Chapitre **3**

Les objets

3.1 LE RETOUR DES STRUCTURES

Les structures du langage C sont utilisables en langage C++ et ont été étendues en leur permettant de contenir des fonctions. Les structures permettent surtout d'introduire la notion de classe définie dans la section 3.2. Une structure en langage C++ (définissable à l'aide du mot-clef `struct`) peut être vue comme une classe sans aucune *encapsulation des données*. Cela signifie que les membres d'une structure ne sont pas protégés et sont modifiables depuis l'extérieur de la structure. Nous rappelons dans cette section comment définir et manipuler des structures.

3.1.1. Déclarer et définir des structures

La *déclaration* d'une structure se fait à l'aide du mot-clef `struct` et est illustrée dans l'exemple suivant. Elle est identique au langage C sauf en ce qui concerne les fonctions que l'on peut maintenant définir comme étant membres d'une structure.

```
1  struct Sidentite
2  {
3    char pcIDEnom[100];
4    char pcIDEprenom[100];
5
6    void IDEaffecter_nom(const char *pcn);
7    void IDEaffecter_prenom(const char *pcn);
```

```
8   void IDEafficher();
9  };
```

Dans cet exemple, nous déclarons une structure (qui permettra d'instancier plusieurs objets) contenant deux champs (des *attributs*) appelés `pcIDEnom` et `pcIDEprenom` et trois fonctions (des *opérations* ou *fonctions membres*) appelées `IDEaffecter_nom`, `IDEaffecter_prenom` et `IDEafficher`. On notera que la déclaration de la structure se termine nécessairement par `;` sous peine d'avoir des erreurs de compilation.

Notez bien qu'il est également possible de déclarer une structure de la façon suivante :

```
1  typedef struct
2  {
3   ...
4  } Sidentite;
```

Bien que le résultat soit le même, la seconde déclaration va impliquer moins de flexibilité dans la déclaration d'objets de cette structure (tout cela vous sera expliqué dans la section suivante).

Une déclaration de structure qui contient des fonctions membres est nécessairement suivie de la *définition* de ces fonctions. À titre d'exemple se trouve ci-dessous la définition de la fonction `IDEafficher`.

```
1  void Sidentite::IDEafficher()
2  {
3   printf("%s %s\n",pcIDEprenom, pcIDEnom);
4  }
```

Nous entrons ici dans le cœur des nouveautés du langage C++ par rapport au langage C. En effet, la définition d'une fonction membre nécessite l'utilisation de ce qui s'appelle **le nom long** de la fonction (ligne 1). Ce nom long se décompose, pour les structures, de la façon suivante :

```
type_retour nom_structure ::nom_fonction(liste_parametres)
```

Sémantiquement ce nom signifie : "la fonction `nom_fonction` qui appartient à la structure `nom_structure`". L'opérateur " :: " s'appelle l'*opérateur de résolution de portée* et définit l'appartenance. Si vous utilisiez le nom de la fonction comme en langage C, que nous appellerons dorénavant le *nom court*, vous définiriez simplement le code d'une fonction qui n'est pas rattachée à une structure. Par conséquent, vous auriez une erreur au linkage de votre programme puisque le linkeur serait incapable de déterminer où se trouve le code de la fonction rattachée à la structure.

La ligne 3 montre que dans une fonction membre on peut référencer directement les attributs de la structure. Cela veut dire que lorsque ces fonctions seront appelées sur des objets ce sont les attributs de ces objets qui seront utilisés.

3.1.2. Utiliser des structures

L'utilisation d'une structure implique la création de variables de ce type, qui seront les objets manipulés. En langage C++ il existe deux façons de déclarer des objets d'une structure comme indiqué dans le code suivant et pour l'exemple de la section précédente.

```
1  void main()
2  {
3   Sidentite IDEclient1;
4   struct Sidentite IDEclient2;
5
6   IDEclient1.IDEaffecter_nom("Bon");
7   IDEclient1.IDEaffecter_prenom("Jean");
8   IDEclient1.IDEafficher();
9   IDEclient2.IDEaffecter_nom("Liguili");
10  IDEclient2.IDEaffecter_prenom("Guy");
11  IDEclient2.IDEafficher();
12 }
```

Les lignes 3 et 4 montrent que la présence du mot-clef `struct` lors de la déclaration d'objets est facultative en langage C++ (en fait, le mot-clef `struct` est toléré pour des raisons de compatibilité avec le langage C). Elle devient interdite si vous avez déclaré votre structure en utilisant le mot-clef `typedef` comme indiqué dans la section précédente.

Les lignes 6 à 11 illustrent comment appeler les fonctions de la structure sur les objets créés. Ainsi, la ligne 6 appelle la fonction `IDEaffecter_nom` sur l'objet `IDEclient1`, ce qui va impliquer que son attribut `pcIDEnom` sera égal à la chaîne "Bon". On retrouve ici la notation pointée des structures du langage C. Il est possible de manipuler directement les chaînes sans passer par les fonctions définies dans la structure `Sidentite`, par exemple en écrivant dans la fonction `main` :

```
strcpy(IDEclient1.pcIDEnom,"Bon");
```

ce qui aurait le même résultat que la ligne 6. On constate ici que l'encapsulation des données n'est absolument pas possible en utilisant les structures : n'importe qui peut faire n'importe quoi sur les attributs de vos structures.

Il est également possible de réaliser directement une affectation entre deux objets d'une même structure. Par exemple, on pourrait très bien écrire :

```
IDEclient2=IDEclient1;
```

ce qui provoquerait la recopie membre à membre des attributs de `IDEclient1` dans `IDEclient2`.

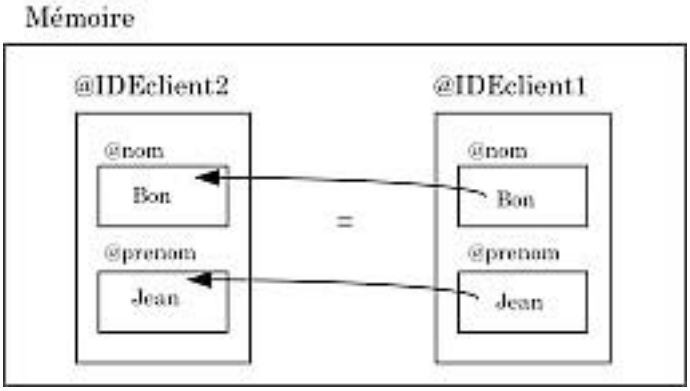


FIG. 3.1: Affectation et recopie membre à membre

La recopie d'un objet d'une structure dans un autre est réalisée comme indiqué sur la figure 3.1 par une recopie des attributs un par un, en aveugle. Cela peut être à l'origine de bugs puisque si un des attributs est un pointeur alors ce qui va être recopié est l'adresse de la zone pointée et non pas la valeur en elle-même. Ce problème sera souligné de nouveau et illustré lors de la présentation des classes. Il est important de noter que sur l'exemple précédent, puisque les attributs de `IDEclient1` et `IDEclient2` sont déclarés comme des tableaux, la recopie membre à membre ne provoquerait pas ce problème bien qu'un tableau soit représenté par l'adresse mémoire de son premier élément.

3.2 LES CLASSES

La structure permet de faire cohabiter des données et des opérations gérant ces données. L'inconvénient est, notamment, que le contenu de la structure reste accessible à tous. Le langage C++ introduit, comme tout langage orienté objets, la notion de classe. Celle-ci généralise et remplace, fonctionnellement parlant, les structures. Ainsi, lorsque vous programmez en C++ vous aurez tendance à ne créer que des classes bien qu'il vous soit syntaxiquement possible de créer des structures. Les classes sont tellement plus riches !

3.2.1. Déclarer et définir des classes

La déclaration et la définition d'une classe sont très simples et reprennent le même schéma que pour la déclaration et la définition d'une structure sauf qu'au lieu du

mot-clef `struct` on utilise le mot-clef `class`. Reprenons l'exemple de la structure `Sidentite` de la section 3.1 que nous transformons maintenant en classe. Nous mettrons toujours la déclaration de la classe dans son interface (ici, le fichier `Cidentite.h`) et sa définition dans le corps (ici, le fichier `Cidentite.cpp`). Voici le contenu du fichier `Cidentite.h`.

```
1  class Cidentite
2  {
3   private :
4    char pcIDEnom[100];
5    char pcIDEprenom[100];
6
7   public :
8    void IDEaffecter_nom(const char *pcn);
9    void IDEaffecter_prenom(const char *pcn);
10   void IDEafficher();
11 };
```

Vous remarquerez que la déclaration de la classe `Cidentite` est vraiment similaire à la déclaration de la structure `Sidentite` sauf en un point précis ici : la présence des mots-clefs `private` et `public`. Ces mots-clefs, que l'on appelle des contrôles d'accès, précisent l'accès des membres de la classe vis-à-vis de *l'extérieur de la classe*[1]. On distingue trois catégories de membres en fonction de leur contrôle d'accès :

– *Les membres publiques.* Le début d'une zone où les déclarations sont publiques est signalé par l'opérateur `public` suivi de " :". Sauf spécification contraire, les déclarations qui suivent ce modificateur seront accessibles avec la notation pointée (ou fléchée) des structures par n'importe quelle fonction ou méthode d'une autre classe. En résumé, on a accès à un membre public en interne (au sein d'une méthode de la même classe) et en externe (au sein d'une fonction n'appartenant pas à la classe).

– *Les membres privés.* Le début d'une zone où les déclarations sont privées est signalé par l'opérateur `private` suivi de " :". Sauf spécification contraire, les déclarations qui suivent ce modificateur ne seront pas accessibles avec la notation pointée (ou fléchée) des structures par n'importe quelle fonction ou méthode d'une autre classe. En résumé, on a accès à un membre privé en interne (au sein d'une méthode de la même classe) mais pas en externe (au sein d'une fonction n'appartenant pas à la classe).

– *Les membres protégés.* Le début d'une zone où les déclarations sont protégées est signalé par l'opérateur `protected` suivi de " :". Un membre protégé subit les

1. On appelle *extérieur* d'une classe `A` toute fonction, appartenant à une autre classe ou non, qui n'est pas membre de cette classe

mêmes restrictions d'accès qu'un membre privé sauf en ce qui concerne l'héritage (voir au chapitre 7) où un membre protégé reste accessible dans une classe héritée ce qui n'est pas le cas d'un membre privé.

Dans l'exemple de la classe `Cidentite` ci-dessus on remarque qu'il y a une première zone de membres privés suivie d'une zone de membres publiques. Ainsi, nous avons encapsulé les deux attributs `pcIDEnom` et `pcIDEprenom` tandis que les méthodes restent accessibles de l'extérieur de la classe. **Il s'agit là d'un comportement qu'il faut tout le temps suivre pour s'assurer d'un développement propre**.

> Sauf cas de force majeure, vous devez toujours déclarer les attributs de vos classes avec le contrôle d'accès privé ou protégé. Les fonctions membres seront quant à elles le plus souvent possibles avec le contrôle d'accès public.

Lorsque vous écrivez une classe vous devez être "paranoïaque" en imaginant mille et une situations au cours desquelles vos attributs sont modifiés sans contrôle aucun (pas de possibilité de vérifier des préconditions ou des postconditions sur ces attributs, perte de cohérence des objets...). Le bon réflexe à avoir est de se dire : "Je mets mes attributs privés et je propose des accesseurs en lecture et en modification pour permettre à mes utilisateurs de consulter ou de modifier ces attributs. Comme ça, je pourrai mettre des préconditions si nécessaire et éviter que les attributs contiennent des valeurs impossibles." Cela va tout à fait dans le sens de la règle d'encapsulation des données (section 1.4.1.c) et de la règle de protection modulaire (section 1.4.2.a, bien que dans cette section nous insistions principalement sur la protection par exceptions). Par ailleurs, encapsuler les attributs d'une classe et mettre en place des accesseurs c'est aussi favoriser la compatibilité ascendante de cette classe. C'est-à-dire que lorsque vous en développerez une nouvelle version plus récente vous limiterez les risques d'avoir à reprendre tout le programme parce que vous avez changé le nom d'un attribut ! Imaginez qu'un attribut change de nom d'une version à une autre et que vous l'ayez laissé en public : il y a des risques pour que vous deviez modifier en partie le reste du programme pour propager le changement de nom. Cela ne risque pas d'arriver si vous avez encapsulé vos attributs.

Il est possible, *mais fortement déconseillé*, de mélanger à volonté les sections `private`, `public` et `protected` au sein d'une classe. En effet, ces contrôles d'accès ne marquent que le début d'une nouvelle zone et la fin de la précédente. D'un point de vue génie logiciel, il est fortement conseillé de décomposer la déclaration d'une classe comme indiqué dans la figure 3.2.

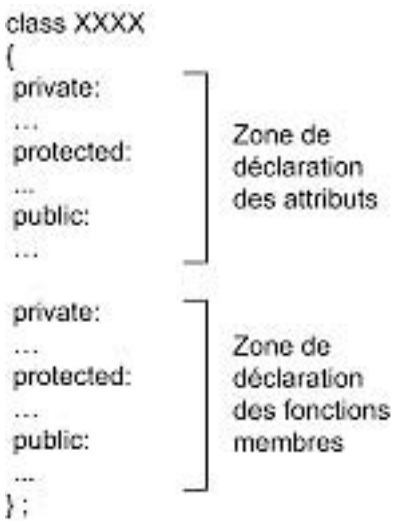


FIG. 3.2: Structure d'une classe déclarée proprement

Il est préférable de d'abord déclarer tous les attributs en utilisant les contrôles d'accès selon vos besoins (n'oubliez pas que sauf cas de force majeure vous devez toujours les déclarer en privé !). Ensuite vous déclarez les *constructeurs et destructeurs* (cf. section 3.4) en public et puis les fonctions membres (généralement en public bien que ce ne soit pas une obligation d'un point de vue génie logiciel). Vous trouverez deux exemples de classes déclarées dans les annexes E et F, avec les commentaires adéquats et une structuration propre. Commencez dès à présent à les lire et essayez d'en comprendre un maximum.

3.2.2. Utiliser des classes

Pour utiliser des classes, il faut tout d'abord créer des objets ! La création d'objets d'une classe, par exemple la classe `Cidentite`, est similaire à la déclaration d'objets d'une structure. Reprenons la fonction `main` de la section 3.1.2. que nous adaptons à l'utilisation de classes.

```
1  void main()
2  {
3   Cidentite IDEclient1;
4   Cidentite IDEclient2;
5
6   IDEclient1.IDEaffecter_nom("Bon");
7   IDEclient1.IDEaffecter_prenom("Jean");
8   IDEclient1.IDEafficher();
9   IDEclient2.IDEaffecter_nom("Liguili");
```

```
10  IDEclient2.IDEaffecter_prenom("Guy");
11  IDEclient2.IDEafficher();
12 }
```

Vous remarquez que seules les lignes 3 et 4 ont changé. En fait, nous verrons dans la section 3.4 qu'il est possible de changer encore les lignes 3 et 4 dans le cas d'objets pour préciser la façon dont ils seront initialisés. Cela sera détaillé dans la section sur les constructeurs et destructeurs.
Il est important de noter ici qu'à la différence des structures il n'est pas possible dans la fonction `main` d'écrire, par exemple :

```
strcpy(IDEclient1.pcIDEnom,"Bon");
```

En effet, l'attribut `pcIDEnom` est déclaré en privé dans la classe `Cidentite` ce qui interdit son accès externe. Autrement dit, pour une fonction qui n'appartient pas à cette classe (c'est le cas de la fonction `main`) il n'est pas possible de référencer cet attribut puisqu'il n'est pas public. Essayez, et vous verrez que vous obtenez une erreur de compilation.

3.2.3. Affecter un objet d'une classe dans un autre objet de la même classe : enjeux et dangers

Bien que ce ne soit pas écrit dans le sommaire ni dans l'introduction cette section est certainement l'une des plus importantes de cet ouvrage.

Pourquoi ? Parce que les classe offrent beaucoup de confort dans leur utilisation et qu'à un moment, si vous ne faites pas attention, vous risquez d'écrire certaines lignes de code lourdes de conséquences. En suivant l'exemple donné dans la section précédente, supposons que vous réalisiez l'affectation entre deux objets d'une même classe.

```
IDEclient2=IDEclient1;
```

Et alors que se passe-t-il ? Et bien, comme pour les structures une recopie membre à membre va être réalisée par le compilateur étant donnée la déclaration de la classe `Cidentite` que vous avez réalisée (cf. section 3.1.1.).

Supposons que cette classe soit maintenant définie comme suit :

```
1  class Cidentite
2  {
3   private :
4     char *pcIDEnom;
5     char *pcIDEprenom;
6
7   public :
8     void IDEaffecter_nom(const char *pcn);
```

```
9    void IDEaffecter_prenom(const char *pcn);
10   void IDEafficher();
11 };
```

Vous remarquerez les attributs qui ont changé de statut et sont passés du statut de tableaux à celui de pointeurs. L'affectation de `IDEclient1` dans `IDEclient2` va provoquer le comportement présenté dans la figure 3.3 et dont le résultat est donné dans la figure 3.4.

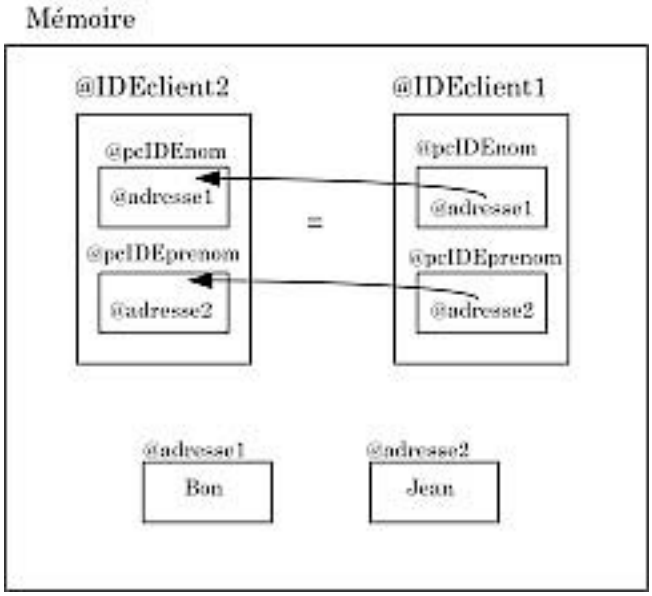


FIG. 3.3: Affectation et recopie membre à membre dans le cas d'attributs de type pointeur

Vous constatez qu'au final les deux objets `IDEclient1` et `IDEclient2` ont des attributs qui pointent sur les mêmes zones mémoires puisqu'il y a eu recopie membre à membre. Conclusion ? Supposez que vous désallouiez (fonction `free` par exemple) les attributs de l'objet `IDEclient1`... le résultat est donné dans la figure 3.5.

Suite à la désallocation, les attributs de `IDEclient1` ne pointent plus sur une zone mémoire allouée tandis que ceux de `IDEclient2` pointent encore sur les adresses précédemment allouées qui, maintenant, ne contiennent plus forcément les valeurs d'avant (les zones mémoires correspondantes ont peut être été réallouées à d'autres objets d'autres programmes par le système d'exploitation). Donc si vous manipulez par la suite les attributs de l'objet `IDEclient2` vous allez avoir des bugs à l'exécution.

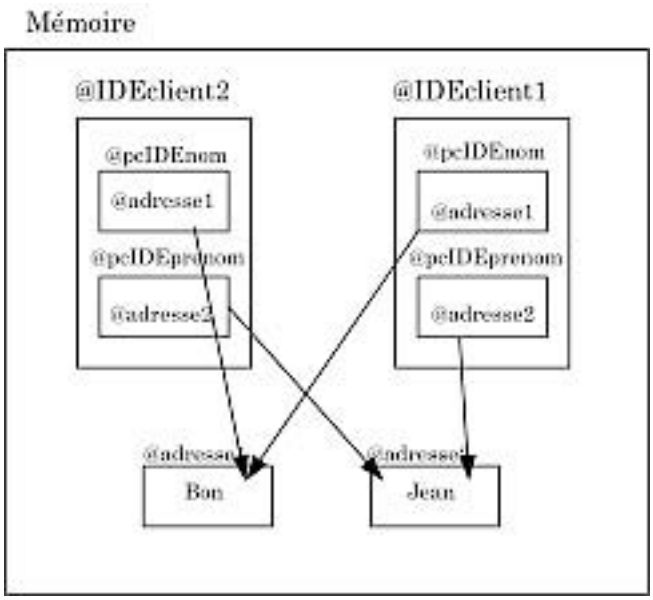


FIG. 3.4: Résultat de l'affectation

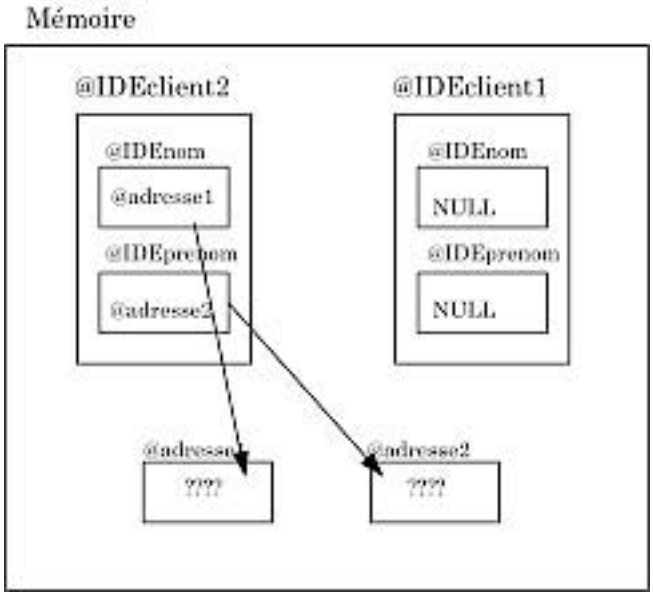


FIG. 3.5: Résultat de la désallocation des attributs `IDEclient1`

Plus précisement des plantages aléatoires, c'est-à-dire des arrêts brutaux de votre programme à des endroits qui suivent, mais non connus, l'affectation délictueuse. En réalité, lorsque vous réalisez l'affectation `IDEclient2=IDEclient1` vous voudriez plutôt avoir le résultat donné dans la figure 3.6.

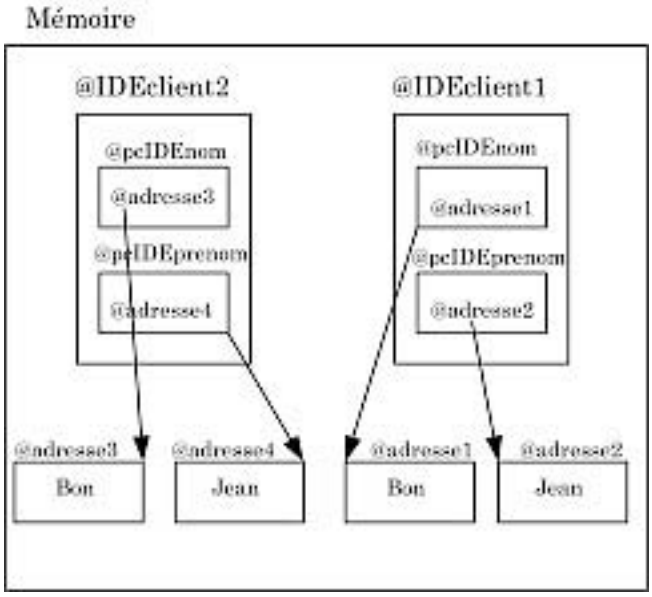


FIG. 3.6: Affectation correcte des objets

De cet exemple on peut en tirer la conclusion suivante.

> La recopie d'objets d'une classe comportant au moins un attribut dynamique (de type pointeur) ne doit pas être réalisée par les opérateurs par défaut qui recopient membre à membre.

Cela inclut l'opérateur d'affectation ainsi que le constructeur de recopie que nous verrons plus tard. La conclusion, **fondamentale**, est donnée ci-dessous.

> Toute classe contenant un ou plusieurs attributs dynamiques (de type pointeur) doit contenir un constructeur de recopie et une surcharge de l'opérateur d'affectation =.

La classe `Cliste` de l'annexe E illustre bien cela. Notez que, d'un point de vue technique vous verrez dans la section 3.4 comment définir un constructeur de recopie et dans le chapitre 8 comment surcharger l'opérateur d'affectation. En tout état de cause, vous devez impérativement avoir le réflexe de définir ces deux fonctions membres dans une classe contenant un ou plusieurs attributs dynamiques.

3.3 VARIABLES ET ATTRIBUTS STATIQUES

Une classe est composée de données et de fonctions membres. D'un point de vue théorique, son instanciation donne lieu à une structure en mémoire, l'objet, contenant les données et fonctions membres de cette classe. En pratique, les méthodes sont partagées entre les objets d'une même classe. Il n'est parfois pas souhaitable de recréer un membre pour chaque objet de la classe. Cela peut être utile, par exemple, si l'on souhaite connaître le nombre de fois où une méthode est appelée, ou bien pour partager des variables entre tous les objets d'une même classe.

3.3.1. Variables statiques n'appartenant pas à une classe

Il est possible de déclarer qu'une variable locale à une fonction est statique en faisant précéder la déclaration de cette variable du mot-clef `static` :

```
static nom_type nom_variable;
```

Par exemple, `static int iV`. L'emploi de ce mot-clef signale au compilateur que la variable qui suit est à créer une fois pour toutes, et qu'elle doit être commune à tous les appels de la fonction qui la définit. Elle est utilisée pour conserver des valeurs localement à une fonction et ce entre deux appels. *En quelque sorte, une variable locale statique peut être vue comme une variable globale mais qui ne peut être référencée qu'à l'intérieur de la fonction qui la déclare*. Par ailleurs, une variable statique est automatiquement initialisée à 0 lors de sa création. Regardons de plus près l'exemple suivant.

```
1  #include <stdio.h>
2
3  void f()
4  {
5   static int iV=0;
6
7   iV++;
8   printf("Valeur de la variable iV : %d\n",iV);
9  }
10
11 void main()
12 {
```

```
13   int iBoucle;
14
15   for (iBoucle=0;iBoucle<5;iBoucle++)
16           f();
17 }
```

D'après vous quel est le résultat de l'affichage réalisé dans la fonction `f` ? Et bien, tout simplement l'affichage suivant.

```
Valeur de la variable iV : 1
Valeur de la variable iV : 2
Valeur de la variable iV : 3
Valeur de la variable iV : 4
Valeur de la variable iV : 5
```

... et ce en dépit de la ligne 5 à laquelle la variable `iV` est initialisée à 0. En effet, cette initialisation s'appelle très précisemment une **initialisation lors d'une déclaration**, c'est-à-dire une initialisation réalisée lors de la déclaration de la variable. Or comme cette variable est statique, elle n'est déclarée (et donc initialisée) qu'une seule fois, au premier appel de la fonction `f`. D'un appel à l'autre, la variable `iV` conserve donc sa valeur.

Par contre, notez bien que si vous n'aviez pas fait une initialisation lors de la déclaration de `iV` vous n'auriez pas obtenu le même résultat. Ainsi, si vous écrivez :

```
1  #include <stdio.h>
2
3  void f()
4  {
5   static int iV;
6   iV=0;
7  ...
```

...la variable `iV` sera systématiquement réinitialisée à 0 à chaque appel à la fonction `f`.

3.3.2. Attributs statiques

Comme pour les variables, il est possible de déclarer que des attributs de classes soient statiques. La syntaxe est la même que pour les variables : il suffit de faire précéder la déclaration de l'attribut par le mot-clef `static`. Cela signale au compilateur que l'attribut qui suit est à créer une fois pour toutes, et qu'il doit être partagé entre toutes les instances de la classe dans laquelle il est défini. C'est une sorte de variable globale aux objets d'une classe. *Ce type d'attribut ne peut pas être initialisé lors de la déclaration de la classe, et doit l'être en dehors de la classe, typiquement*

au sein de son corps (fichier `.cpp`*).* Par ailleurs, une variable statique est automatiquement initialisée à 0 lors de sa création, si vous ne précisez pas d'initialisation. Reprenons l'exemple de la classe `Cidentite` vue dans la section 3.2 et supposons qu'elle contienne un attribut statique supplémentaire déclaré comme suit :

```
static int iCompteur;
```

qui permette de compter le nombre d'objets de la classe `Cidentite` créés dans votre programme. Pour initialiser cet attribut il faut ajouter la ligne suivante **en dehors** de la déclaration de la classe (par exemple dans le fichier `Cidentite.cpp`) :

```
int Cidentite::iIDECompteur=0;
```

Notez bien l'utilisation du nom long pour nommer l'attribut : sans l'utilisation de l'opérateur de résolution de portée le compilateur aurait compris que vous vouliez créer une variable globale `iCompteur` initialisée à 0.

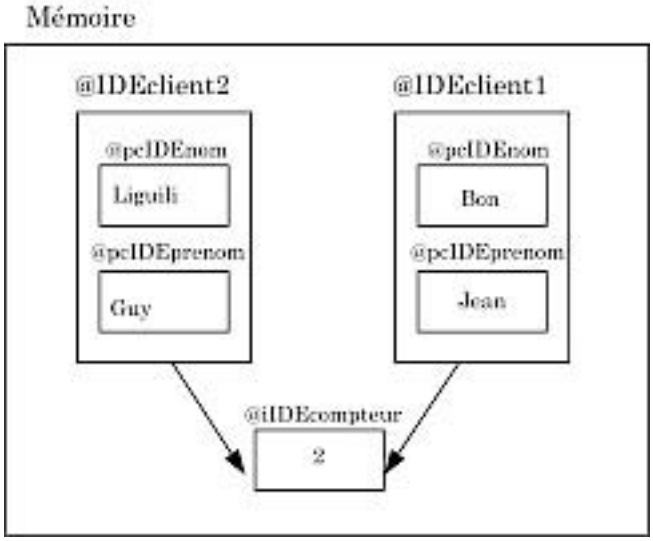


FIG. 3.7: Gestion mémoire des attributs statiques

Dans la figure 3.7 est illustrée la façon dont sont gérés les attributs statiques. Puisqu'un attribut statique est commun à tous les objets d'une même classe il lui est alloué une zone mémoire dans laquelle est stockée sa valeur. Chaque objet d'une classe "pointe" alors vers cette zone mémoire ce qui lui donne accès au contenu de l'attribut statique. Cette organisation implique qu'un attribut statique peut être référencé sans qu'aucun objet de la classe ne soit instancié, comme montré dans le code ci-dessous.

```
1 void main()
2 {
3         printf("Valeur initiale du compteur : %d\n",Cidentite::iIDEcompteur);
4 }
```

L'attribut statique `iIDEcompteur` est simplement référencé par son nom long puisqu'aucun objet de la classe `Cidentite` n'est utilisé pour cela.

3.4 CONSTRUCTEURS ET DESTRUCTEUR

Une nouveauté importante liée aux classes par rapport aux structures est l'apparition de constructeurs et de destructeurs. Ce sont des fonctions membres automatiquement appelées à la naissance et à la mort de l'objet. Cela simplifie grandement l'utilisation des objets par rapport aux structures, pour lesquelles ces opérations devaient être appelées par l'utilisateur. Les constructeurs et les destructeurs, d'après leur nature sont des fonctions qui ne renvoient aucun résultat.

3.4.1. Les constructeurs

Le constructeur est une fonction membre qui a le même nom que la classe dans laquelle il est défini. Son rôle est de réaliser l'initialisation des objets d'une classe lorsque ceux-ci sont créés. Le constructeur est appelé automatiquement juste après la création en mémoire de l'objet (celui-ci doit exister pour qu'une fonction membre puisse être appelée). **Un constructeur sert donc à réaliser une initialisation lors de la déclaration d'un objet**. Par exemple, lorsque vous écrivez :

```
Cidentite IDEv;
```

l'objet `IDEv` est initialisé par l'appel à un constructeur lors de sa déclaration. Cette ligne de code provoque donc deux actions : (i) la création en mémoire de l'objet `IDEv`, (ii) l'appel à un constructeur pour l'initialiser.

> Toute création d'un objet provoque l'appel à un constructeur de sa classe d'appartenance.

On peut définir autant de constructeurs que l'on désire, avec un nombre de paramètres différents. De même, un constructeur doit être déclaré dans une zone `public` pour pouvoir être appelé de l'extérieur de la classe lors de la déclaration des objets. Néanmoins, d'un point de vue technique, il est possible de déclarer des constructeurs

dans une zone `private` ou `protected`. Cela entraîne qu'il est alors impossible de créer des objets initialisés par appel à ces constructeurs, ce qui n'est pas très pratique.

On distingue deux constructeurs particuliers : le *constructeur par défaut* et le *constructeur de recopie*. Ce dernier n'a vraiment de sens que si la classe contient des attributs dynamiques (cf. section 3.2.3. pour une explication). Nous allons maintenant voir plus en détail ces deux constructeurs particuliers.

a) Le constructeur par défaut

Le constructeur par défaut ne prend pas de paramètre. Il est appelé à la déclaration d'un objet dès lors que le compilateur ne dispose d'aucune information sur le constructeur à appeler. Souvent il ne fait rien mais il peut être employé pour affecter des valeurs par défaut aux attributs d'un objet. *Si aucun constructeur par défaut n'est défini pour une classe, un constructeur par défaut (appelé le constructeur par défaut par défaut) qui ne fait rien, est automatiquement attribué par le compilateur à cette classe.* Reprenons l'interface de la classe `Cidentite` et ajoutons un constructeur par défaut.

```
1  class Cidentite
2  {
3   private :
4     char pcIDEnom[100];
5     char pcIDEprenom[100];
6
7   public :
8     Cidentite();
9     void IDEaffecter_nom(const char *pcn);
10    void IDEaffecter_prenom(const char *pcn);
11    void IDEafficher();
12 };
```

Remarquez que le constructeur par défaut, déclaré à la ligne 8, ne prend aucun paramètre et ne possède aucun type de retour, même pas le type `void`, ce qui est normal puisqu'un constructeur ne peut pas retourner de valeur. La définition de ce contructeur se fait dans le corps de la classe (fichier `Cidentite.cpp`) à l'identique de la définition d'une fonction membre, c'est-à-dire en utilisant le nom long du constructeur.

> Notez bien que si pour une classe vous ne définissez pas de constructeur par défaut mais au moins un constructeur prenant des arguments alors le compilateur refusera d'utiliser son constructeur par défaut par défaut en cas de besoin. Vous

aurez donc un message d'erreur à la compilation sur la ligne où vous déclarez un objet initialisé par ce constructeur.

b) Le constructeur de recopie

Le constructeur de recopie sert à initialiser un objet par recopie à partir d'un objet existant. Regardez l'exemple ci-dessous.

```
1  void main()
2  {
3   Cidentite IDEclient1;
4   Cidentite IDEclient2(IDEclient1);
5   ...
6  }
```

Sur la ligne 3, un objet nommé `IDEclient1` est créé et initialisé par appel au constructeur par défaut de la classe `Cidentite`. Sur la ligne 4, un objet nommé `IDEclient2` est créé et initialisé par recopie du contenu de l'objet `IDEclient1` et grâce à l'appel du constructeur de recopie de la classe `Cidentite`.

Un constructeur de recopie est un constructeur qui prend en argument un objet de la classe passé par référence. En voici l'interface dans le cadre de la classe `Cidentite`.

```
1  class Cidentite
2  {
3   private :
4    char pcIDEnom[100];
5    char pcIDEprenom[100];
6
7   public :
8    Cidentite();
9    Cidentite(Cidentite &IDEarg);
10   void IDEaffecter_nom(const char *pcn);
11   void IDEaffecter_prenom(const char *pcn);
12   void IDEafficher();
13 };
```

Le constructeur de recopie est déclaré à la ligne 9. La définition de ce constructeur se fait dans le corps de la classe (fichier `Cidentite.cpp`) à l'identique de la définition d'une fonction membre, c'est-à-dire en utilisant le nom long du constructeur.

```
1  Cidentite : :Cidentite(Cidentite &IDEarg)
2  {
```

```
3   pcIDEnom=IDEarg.pcIDEnom ;
4   pcIDEprenom=IDEarg.pcIDEnom ;
5 }
```

Notez bien que le constructeur de recopie a tout à fait le droit de référencer les attributs privés de l'objet `IDEarg` puisqu'ils sont de la même classe : *la portée des contrôles d'accès est la classe*, pas l'objet.

Dans un constructeur de recopie, le passage de l'argument par référence est obligatoire. En effet, si vous réalisiez un passage par adresse vous n'auriez plus un constructeur de recopie mais un constructeur à un argument prenant une adresse. Il faut donc passer un objet pour satisfaire aux besoins de la recopie. Mais si on pouvait réaliser un passage par valeur on serait confronté à un problème cornélien ! Supposez que l'on puisse écrire dans la classe `Cidentite` la ligne 6 du code suivant.

```
1 class Cidentite
2 {
3   ...
4   public :
5     Cidentite() ;
6     Cidentite(Cidentite IDEarg) ;
7 ...
8 };
```

Que se passe-t-il à la création d'un objet initialisé par recopie (reprenons l'exemple du début de section) ? L'objet `IDEclient1` de notre exemple est maintenant passé par valeur, ce qui veut dire que le compilateur crée un objet temporaire de la classe `Cidentite` qui sera initialisé par recopie à partir de `IDEclient1`. Cet objet temporaire sera passé au constructeur de recopie pour qu'il initialise `IDEclient2`. Oui mais nous venons d'écrire que "le compilateur va créer un objet temporaire initialisé par le constructeur de recopie", ce qui provoque un nouvel appel au constructeur que nous cherchons déjà à appeler. Donc le compilateur, pour initialiser l'objet temporaire, va créer un second objet temporaire puisque le constructeur de recopie demande un passage par valeur de l'argument... En conclusion, autoriser un passage par valeur provoque une boucle infinie : le constructeur de recopie serait appelé indéfiniment sans jamais avoir le temps de s'exécuter ! Donc, il est interdit en langage C++ de faire du passage par valeur dans un constructeur de recopie et seul le passage par référence est possible. Essayez et vous obtiendrez une erreur de compilation.

Le passage par référence de l'argument implique qu'en théorie l'argument peut être modifié dans le constructeur. Pour empêcher cela, il vous est possible de déclarer que l'argument est de type constant (présence du mot-clef `const` devant le type de l'argument). D'ailleurs, comme par définition un constructeur de recopie ne doit

pas modifier son argument (il ne fait que de la recopie d'attributs) il est même très fréquent de déclarer que l'argument est constant en pratique !

Le langage C++ dispose d'un **constructeur de recopie par défaut** qui est utilisé dans une classe dans laquelle le programmeur n'a pas défini son propre constructeur de recopie. Ainsi, si dans l'exemple de la classe `Cidentite` nous n'avions pas déclaré un tel constructeur, le compilateur aurait automatiquement ajouté le constructeur de recopie par défaut à la compilation. Mais que fait ce constructeur de recopie par défaut ? Tout simplement une **recopie membre à membre des attributs de la classe**. Et revoilà cette fameuse recopie membre à membre dont nous avons déjà parlé dans la section 3.2.3. dans le cadre de l'affectation de deux objets d'une même classe. Les phénomènes présentés dans cette section sont toujours d'actualité : si vous avez une classe qui dispose d'attributs dynamiques alors vous devez ajouter votre constructeur de recopie sous peine d'avoir les mêmes problèmes que ceux énoncés dans la section 3.2.3. puisque vous allez recopier en aveugle les valeurs des attributs d'un objet dans un autre. En plus, il est très facile de faire apparaître des bugs suite à l'oubli d'un constructeur de recopie dans une classe possédant des attributs dynamiques. Regardez l'exemple suivant pour vous en convaincre.

```
1  class Cidentite
2  {
3   private :
4    char *pcIDEnom ;
5    char *pcIDEprenom ;
6
7   public :
8    Cidentite() ;
9    void IDEaffecter_nom(const char *pcn) ;
10   void IDEaffecter_prenom(const char *pcn) ;
11   void IDEafficher() ;
12 } ;
13 ...
14 void main()
15 {
16  Cidentite IDEclient1 ;
17  Cidentite IDEclient2(IDEclient1) ;
18  ...
19 }
```

Vous remarquez que nous avons changé la définition des deux attributs pour qu'ils soient de type dynamique. Le bug est introduit à la ligne 17 lors de l'appel au constructeur pour initiliser l'objet `IDEclient2`. Tout serait trop simple si à l'exécution votre programme plantait sur cette ligne-là : malheureusement le bug que vous avez introduit peut potentiellement faire planter votre programme et bien plus loin que la ligne 17. C'est aussi cela qui rend la détection du bug difficile. Alors

autant bien faire les choses dès le départ et avoir les bons réflexes dès l'écriture de vos classes.

> Dans une classe possédant au moins un attribut dynamique (de type pointeur) vous devez impérativement définir votre propre constructeur de recopie. Souvenez-vous que vous devez également définir une surcharge de l'opérateur d'affectation "=".

c) Autres constructeurs

Il est possible de créer autant de constructeurs avec autant d'arguments que l'on veut. Il suffit juste de spécifier quel est le constructeur appelé lors de la déclaration des objets en indiquant après le nom de l'objet les valeurs que l'on utilise pour l'initialiser. À partir de ces valeurs, le compilateur choisit automatiquement le constructeur à appeler. Reprenons l'exemple de la classe `Cidentite`.

```
1  class Cidentite
2  {
3   private :
4     char pcIDEnom[100];
5     char pcIDEprenom[100];
6
7   public :
8     // Constructeurs
9     Cidentite(); // Constructeur par défaut
10    Cidentite(const Cidentite & objet); // Constructeur de recopie
11    Cidentite(char *pcn,char *pcp); // Constructeur à deux arguments
12    // Fonctions membres
13    void IDEaffecter_nom(const char *pcn);
14    void IDEaffecter_prenom(const char *pcn);
15    void IDEafficher();
16 };
```

Trois constructeurs ont été ici définis : le constructeur par défaut (ligne 9), le constructeur de recopie (ligne 10) et un constructeur à deux arguments (ligne 11) qui permet d'initialiser les attributs `pcIDEnom` et `pcIDEprenom` à la création de l'objet. Ainsi, dans les déclarations ci-dessous...

```
1  Cidentite IDEo1; // Appel au constructeur par défaut
2  Cidentite IDEo2(IDEo1); // Appel au constructeur de recopie
3  Cidentite IDEo3("Goutte","Anne"); // Appel au constructeur à deux arguments
```

... trois objets de la classe `Cidentite` sont créés et initialisés avec un constructeur différent pour chacun.

Rappel : la définition d'un constructeur se fait comme pour une fonction membre ordinaire, c'est-à-dire de la façon suivante dans le fichier `.cpp` dans le cas du constructeur de recopie :

```
Cidentite::Cidentite(const Cidentite & objet)
{
...
}
```

d) Initialisation des attributs depuis le constructeur

Il existe en langage C++ une syntaxe simplifiée pour initialiser des attributs d'une classe directement à partir de l'interface d'un constructeur. Pour cela, il suffit de faire suivre la déclaration de l'interface du constructeur par " :" suivi du nom des attributs à initialiser et de leur initialisateur. En voici un exemple.

```
1 Cidentite : :Cidentite(char * pcn, char *pcp) : pcIDEnom(pcn),pcIDEprenom(pcp)
2 {
3  ...
4 }
```

Dans cet exemple, l'appel au constructeur à deux arguments de la classe `Cidentite` provoque l'initialisation de l'attribut `pcIDEnom` par le paramètre `pcn` et l'attribut `pcIDEprenom` par le paramètre `pcp`. Notez que cette initialisation est réalisée avant d'exécuter le code du constructeur. Ce mécanisme n'a guère d'intérêt pour l'initilisation d'attributs d'un type de base. En effet, le code précédent est équivalent au code ci-dessous (dans le sens où il provoque le même résultat).

```
1 Cidentite : :Cidentite(char * pcn, char *pcp)
2 {
3  pcIDEnom=pcn ;
4  pcIDEprenom=pcp ;
5  ...
6 }
```

Néanmoins ce mécanisme devient nécessaire pour l'initialisation des attributs qui sont des objets d'une autre classe, puisqu'il permet de déterminer les constructeurs utilisés pour initialiser ces attributs. Pour illustrer cela, supposons maintenant que la classe `Cidentite` possède un attribut nommé `adresse` et qui soit d'une classe `Cadresse` définie par ailleurs. On peut écrire le code suivant...

```
1 Cidentite : :Cidentite(char * pcn, char *pcp) : adresse(pcn,pcp)
2 {
3  ...
4 }
```

... qui provoque l'appel à un constructeur à deux arguments de type `char` de la classe `Cadresse` pour initialiser l'attribut `adresse` (évidemment il est nécessaire qu'un tel constructeur existe pour que ce code puisse compiler sans erreur). Ainsi, cette syntaxe permet de préciser simplement quel constructeur appeler avec quels arguments.

3.4.2. Le destructeur

Le destructeur est une fonction qui est appelée automatiquement à la destruction de l'objet, juste avant la désallocation de l'objet en mémoire. Cette fonction ne prend jamais d'argument et porte le nom de la classe précédé du symbole " $\sim$ ".

```
1  class Cidentite
2  {
3   private :
4     char pcIDEnom[100];
5     char pcIDEprenom[100];
6
7   public :
8     // Constructeurs
9     ...
10    // Destructeur
11    ~Cidentite();
12    // Fonctions membres
13    ...
14 };
```

Le destructeur est essentiellement utilisé lorsque la classe possède des attributs de type pointeurs, c'est-à-dire des attributs qui ont été alloués auparavant et qui doivent être désalloués avant la destruction de l'objet. Les instructions de désallocation (`free`, `delete`) doivent être alors utilisées dans le destructeur. Dans le cas d'une classe ne possédant pas d'attributs dynamiques, le corps du destructeur est très souvent vide. Notez que si vous ne mettez pas de destructeur dans votre classe le compilateur en mettra une version par défaut qui est vide.

3.5 GESTION D'OBJETS DYNAMIQUES : LES OPÉRATEURS `new` ET `delete`

Le langage C++ introduit deux nouveaux opérateurs de gestion mémoire : et . Le premier effectue l'allocation d'une zone mémoire et le second réalise sa libération. Ces deux opérateurs étendent les fonctions et du langage C. En effet, celles-ci n'ont pas été prévues pour l'allocation d'objets de classes et notamment l'appel aux constructeurs/destructeurs. N'oubliez pas qu'à chaque création d'objet il y a forcément un constructeur qui est appelé, or la fonction `malloc` a été écrite pour le langage C... dans lequel la notion de constructeur n'existait pas. C'est pour cette raison que l'opérateur `new` a été introduit : pour allouer des objets **et** appeler des constructeurs pour les initialiser. Le même raisonnement vaut pour l'opérateur `delete` vis-à-vis de la fonction `free`. *Notez que les opérateurs `new` et `delete` remplacent complètement les fonctions `malloc` et `free` (fonctionnellement parlant) puisqu'ils peuvent être également utilisés pour allouer des variables autres que d'une classe*.

Ces opérateurs ont la faculté de déterminer automatiquement le type de l'objet à allouer, et donc de renvoyer un pointeur ayant ce type. Ainsi, on n'est plus obligé d'utiliser la conversion explicite (cast) comme avec `malloc` ou `free`. Ils déterminent également de manière automatique la taille des éléments à allouer ou à libérer. L'instruction `malloc` requiert la taille mémoire à allouer, l'opérateur `new` ne nécessite que le nombre d'éléments à allouer.

Bien que les instructions `malloc`/`free` et `new`/`delete` puissent cohabiter, il est recommandé de ne pas les mélanger : une allocation réalisée avec `malloc` doit être libérée par `free`, tandis qu'une allocation réalisée par `new` doit être libérée par `delete`.

3.5.1. Allocation d'un élément unique ou d'un tableau d'éléments

L'opérateur `new` permet l'allocation d'un élément unique que ce soit un objet d'une classe ou un objet (appelé variable en règle générale) d'un type de base, d'une structure... Dans le cas d'un objet, un constructeur de sa classe d'appartenance est appelé après la réalisation de l'allocation mémoire.

`new=malloc` + appel au destructeur

Pour utiliser l'opérateur `new`, il suffit d'utiliser la syntaxe suivante.

```
ptr_var = new type_var;
```

Un exemple vous est proposé ci-dessous. Notez que dans cet exemple, la ligne 6 provoque l'allocation mémoire d'un objet de la classe `Cidentite` puis l'appel au constructeur par défaut de cette classe pour initialiser l'objet `pIDEclient1`.

```
1 void main()
2 {
```

```
3   Cidentite * pIDEclient1;
4
5   // Allocation d'un objet unique
6   pIDEclient1 = new Cidentite;
7   ...
8  }
```

Il est également possible de préciser, lors de l'appel à `new` quel constructeur vous souhaitez appeler pour initialiser votre objet. Pour cela, il suffit d'utiliser une syntaxe légèrement modifiée en faisant suivre `type_var` des valeurs/objets, entre parenthèses, qui vont servir au compilateur à identifier le constructeur à appeler.

```
1  void main()
2  {
3   Cidentite * pIDEclient1;
4   Cidentite * pIDEclient2;
5
6   // Allocation de deux objets uniques
7   pIDEclient1 = new Cidentite;
8   pIDEclient2 = new Cidentite(*pIDEclient1);
9   ...
10 }
```

Dans l'exemple ci-dessus, la ligne 7 provoque l'initialisation de l'objet `pIDEclient1` par l'appel au constructeur par défaut puisqu'aucune valeur/objet n'est spécifiée à l'opérateur `new`. La ligne 8 provoque un traitement un peu différent puisque, après l'allocation mémoire réalisée par `new`, le compilateur doit appeler un constructeur de la classe `Cidentite` prenant en paramètre un objet de cette classe (ici, l'objet `*pIDEclient1`). Il ne peut s'agir que du constructeur de recopie.

L'opérateur `new` permet également d'allouer des tableaux d'éléments d'un type de base ou d'une classe. Comme pour l'allocation d'un élément unique, *si ceux-ci sont d'une classe il y aura nécessairement appel à un constructeur pour chacun des éléments du tableau*. La syntaxe est la suivante.

```
ptr_var = new type_var[nb_elements];
```

Il est important de noter **qu'il n'est pas possible de choisir le constructeur utilisé pour les objets d'un tableau** comme nous l'avons vu ci-dessus : chacun des objets de votre tableau sera nécessairement initialisé par appel au constructeur par défaut de la classe.

Dans la librairie standard du langage C est fourni un jeu de fonctions pour l'allocation mémoire : `malloc`, `calloc`, `realloc`... L'opérateur `new` du langage C++ vient étendre ces deux premières puisqu'il permet l'allocation de nouveaux pointeurs.

Ceci étant dit, *aucun opérateur ne vient étendre la fonction* `realloc` ce qui peut être gênant dès que vous avez une liste d'objets de taille variable à gérer. En effet, la fonction `realloc` permet de changer la taille d'un pointeur en lui allouant plus ou moins de mémoire. Comme pour la fonction `malloc`, cette fonction n'appelle pas le constructeur/destructeur sur les objets alloués/désalloués suite au changement de taille du pointeur. Comment faire alors pour gérer des listes d'objets de tailles variables ? Une solution consiste à passer par un double pointeur sur une classe, le premier niveau de pointeur étant géré par la fonction `realloc` tandis que le second est géré à l'aide de l'opérateur `new`. L'exemple ci-dessous montre comment allouer et réallouer une liste d'objets de la classe `Cidentite`.

```
1  #include "Cidentite.h"
2  #include <stdlib.h>
3
4  void main()
5  {
6   Cidentite ** pIDEclient1;
7   int iBoucle;
8
9   // Allocation d'un tableau de pointeurs
10  pIDEclient1 = (Cidentite **) malloc(10*sizeof(Cidentite *));
11  for (iBoucle=0;iBoucle<10;iBoucle++)
12          pIDEclient1[iBoucle] = new Cidentite;
13  // Un tableau de 10 objets Cidentite est alloué et initialisé
14
15  pIDEclient1 = (Cidentite **) realloc(pIDEclient1, 15*sizeof(Cidentite *));
16  for (iBoucle=10;iBoucle<15;iBoucle++)
17          pIDEclient1[iBoucle] = new Cidentite;
18  // Le tableau est étendu à 15 objets
19
20  for (iBoucle=13;iBoucle<15;iBoucle++)
21          pIDEclient1[iBoucle] = delete Cidentite;
22  pIDEclient1 = (Cidentite **) realloc(pIDEclient1, 13*sizeof(Cidentite *));
23  // Le tableau est réduit à 13 objets
24 }
```

Dans cet exemple, les lignes 10 à 12 permettent tout d'abord de créer un premier tableau, chaque case du tableau étant une adresse sur un objet de la classe `Cidentite` (ligne 10). Ensuite, pour chacunes de ces cases, un objet est alloué dynamiquement à l'aide de l'opérateur `new` (ligne 12) avec appel au constructeur par défaut. Les lignes 15 à 17 permettent d'étendre la taille du tableau à 15 objets en augmentant le nombre de cases du tableau puis en allouant un objet par case. Enfin, les lignes 20 à 22 réduisent la taille du tableau à 13 objets en *désallouant d'abord* les

objets en position 13 et 14 dans le tableau puis en réduisant son nombre de cases. Notez ici l'utilisation de l'opérateur `delete` dont le fonctionnement est détaillé dans la section suivante.

Note : Cet exemple vous permet également d'imaginer comment faire pour créer un tableau dont chaque élément est initialisé par appel à un constructeur autre que le constructeur par défaut (changez la ligne 12 par exemple en utilisant la syntaxe vue dans cette section pour préciser le constructeur à appeler dans le cas d'un élément unique).

3.5.2. Désallocation d'un élément unique ou d'un tableau d'éléments

L'opérateur `delete` permet de désallouer un pointeur sur un objet ou une variable d'un type de base. Dans le cas d'un objet, le destructeur de sa classe d'appartenance est appelé avant la réalisation de la désallocation mémoire.

`delete`=appel au destructeur + `free`

La syntaxe d'utilisation de cet opérateur sur un élément unique est la suivante.

```
delete type_var;
```

Dans le cas de la désallocation d'un pointeur sur plusieurs objets (tableau simple d'objets), la syntaxe est la suivante.

```
delete [] type_var;
```

```
1  void main()
2  {
3   Cidentite * pIDEclient1;
4
5   pIDEclient1 = new Cidentite[10];
6   // Le tableau est alloué
7   ...
8   pIDEclient1 = delete [] Cidentite;
9   // Le tableau est désalloué
10 }
```

3.6 TOUT SUR LA VIE DES OBJETS : SYNTHÈSE

Les variables (et donc les objets) peuvent être créées de deux manières en langage C++ : la première est une déclaration tandis que la seconde utilise un opérateur d'allocation.

Dans le cas d'une déclaration, la variable est statique ou automatique. Sa durée de vie est définie par l'emplacement de sa déclaration et par sa nature (statique ou

automatique). Dans le cas d'une allocation dynamique, la durée de vie est contrôlée par le programme (dans les limites de la portée maximale de la variable).

3.6.1. Initialisation d'un objet dès sa déclaration

Une variable (et donc un objet) peut être initialisée à sa déclaration de manière explicite. Pour cela on fait suivre le nom de la variable du signe "=" et d'une expression appelée initialisateur (on peut aussi le mettre entre parenthèses plutôt que d'utiliser le signe "="). Dans le cas des objets, ce mécanisme est important car il permet de transmettre des paramètres au constructeur de la classe. Pour cela, la classe de l'objet doit comporter un constructeur compatible avec les types des paramètres passés.

```
1  class Cidentite
2  {
3   ...
4   public :
5      Cidentite(int iP) ;
6  } ;
7
8  void main()
9  {
10  Cidentite IDEclient1=3 ; // Ces deux syntaxes
11  Cidentite IDEclient2(5) ; // sont équivalentes
12  ...
13 }
```

Notez que sur les lignes 10 et 11 nous aurions très bien pu mettre, par exemple, un initialisateur de type `float` : le constructeur à un argument de type `int` serait resté *compatible* et aurait été appelé (via une dégradation numérique de `float` vers `int`). Si jamais le compilateur n'arrive pas à déterminer le constructeur à appeler sur les lignes 10 ou 11, alors vous aurez un message d'erreur à la compilation sur la ligne fautive (et pas à la compilation de la classe `Cidentite`!).

Notez également que si sur la ligne 5, l'argument `iP` était passé par référence il aurait impérativement fallut utiliser le mot-clef `const` sur cette ligne sous peine de ne pas pouvoir réaliser les lignes 10 et 11 (confère section 2.3.1.).

3.6.2. Différentes catégories d'objets

a) Les objets statiques

Un objet statique est créé par une déclaration précédée du mot-clef `static` (cf. section 2.2.2.). Un tel objet est créé avant la première exécution de la première instruction du bloc dans lequel il est déclaré, et il est détruit après la dernière exécution

de ce bloc. Les objets sont détruits dans l'ordre inverse de leur construction (le premier construit est le dernier détruit). Ces objets sont nécessairement initialisés par un appel à un constructeur.

b) Les objets automatiques

Un objet automatique est créé par une déclaration dans un bloc ou dans une fonction. Sa durée de vie est limitée à l'étendue de la fonction ou du bloc concerné. L'objet est automatiquement détruit dès que l'on quitte cette fonction ou ce bloc. Les objets sont détruits dans l'ordre inverse de leur construction (le premier construit est le dernier détruit). Ces objets sont nécessairement initialisés par un appel à un constructeur.

c) Les objets temporaires

Un objet temporaire est créé implicitement par le programme dans certains cas de figure. Vous n'avez aucun contrôle sur ces objets puisque c'est le compilateur qui décide quand les créer et les détruire. Cela présente un inconvénient important : celui de ralentir potentiellement l'exécution de votre programme et d'augmenter l'espace mémoire dont il a besoin. Nous avons déjà vu un exemple dans lequel était créé un objet temporaire : celui du passage d'un argument par valeur à une fonction (cf. annexe C). Dans cet exemple, une copie de la variable `iV1` était transmise à la fonction `f`. Si cette variable est un objet d'une classe alors le compilateur, pour réaliser l'appel à la fonction, va créer un objet temporaire initialisé par le constructeur de recopie. Cet objet temporaire est détruit automatiquement à la sortie de la fonction. Notez ici une source potentielle de bug : imaginez que vous ayez écrit une classe ayant un attribut dynamique (pointeur par exemple) et sans constructeur de recopie. Le passage par valeur à une fonction d'un objet de cette classe peut provoquer les mêmes problèmes que ceux soulevés dans la section 3.2.3.

Voici quelques situations les plus courantes au cours desquelles des objets temporaires sont créés :

(1) Passage par valeur d'un objet d'une classe à une fonction.

(2) Renvoie par valeur d'un objet d'une fonction.

(3) Évaluation d'une expression arithmétique (voir le chapitre sur la surcharge d'opérateurs et de types).

d) Les objets dynamiques

Les objets dynamiques sont les objets déclarés sous forme de pointeurs et initialisés par l'opérateur `new`. Leur durée de vie peut être contrôlée par l'opérateur `delete`. L'appel au constructeur/destructeur est réalisé lors de l'appel à ces opérateurs. Dans le cas de l'appel au constructeur, l'opérateur `new` appel le constructeur par défaut de la classe à moins que des arguments ne soient passés comme indiqué dans la section 3.5.1.

e) Créer des tableaux d'objets

Pour créer un tableau d'objets vous avez deux possibilités : créer un tableau d'objets automatiques ou créer un tableau d'objets alloués dynamiquement.

Dans le premier cas de figure, chaque objet du tableau est initialisé, *sans autre précision de votre part*, par l'appel au constructeur par défaut. Mais vous avez également la possibilité de préciser pour chaque objet quel constructeur doit être appelé.

```
1 void main()
2 {
3  Cidentite pIDEclient1[5];
4  Cidentite pIDEclient2[5]={7,3,pIDEclient1[0]};
5
6  ...
7 }
```

Dans l'exemple ci-dessus, tous les objets du tableau `pIDEclient1` sont initialisés par appel au constructeur par défaut. Cela est différent pour les objets du tableau `pIDEclient2` puisque l'objet en position 0 dans ce tableau sera initialisé avec la valeur 7, celui en position 1 avec la valeur 3 et celui en position 2 par recopie de l'objet `pIDEclient1[0]`. Les objets en positions 3 et 4 sont initialisés par appel au constructeur par défaut puisqu'il n'y a plus de valeurs disponibles après le signe "=". Évidemment, ce mécanisme suppose que le programmeur de la classe `Cidentite` ait inclus les bons constructeurs au sein de sa classe pour que la ligne 4 puisse compiler (notamment, ici, présence d'un constructeur à un argument de type `int` ou compatible).

Il est également possible de créer un tableau d'objets qui soit alloué dynamiquement à l'aide de l'opérateur `new` (cf. section 3.5.1.). Vous noterez qu'il ne vous est pas possible de sélectionner le constructeur à appeler pour chacun des objets.

3.7 COMMENT BIEN ÉCRIRE UNE CLASSE ?

D'un point de vue purement langage, vous pouvez écrire une classe comme bon vous semble. D'un point de vue génie logiciel un certain nombre de règles doivent être respectées pour faciliter la lecture et l'utilisation de vos classes, limiter le nombre de bugs... Vous trouverez deux exemples dans les annexes E et F.

Écrire une classe implique de suivre des normes de présentation. Ces deux annexes vous proposent une norme de rédaction que vous pourrez suivre ou faire évoluer selon vos besoins.
Bien écrire une classe implique de suivre une démarche systémique dans l'étape de conception. Voici un petit mémento à appliquer dès que vous aurez une classe à écrire :

(1) Commencez par identifier les attributs définissant votre classe. N'en mettez pas trop, quitte à créer d'autres classes et à inclure comme attributs des objets de ces classes. Réfléchissez au type adéquat pour chacun des attributs. De même, mettez vos attributs avec le contrôle d'accès privé ou protégé (sauf "raison d'État" contraire).

(2) Passez aux constructeurs et posez-vous les questions suivantes : Faut-il un constructeur par défaut ? Avez-vous des attributs dynamiques (auquel cas prévoyez un constructeur de recopie et une surcharge de l'opérateur =) ? Avez-vous besoin d'autres constructeurs (pour réaliser des conversions, initialiser plus rapidement vos objets...) ? Ces constructeurs sont à prévoir avec le contrôle d'accès public.

(3) Posez-vous la question de savoir s'il faut un destructeur autre que le destructeur par défaut (qui ne fait rien). Avez-vous des attributs dynamiques ? Si oui, il faut vraisemblablement prévoir leur désallocation dans le destructeur sous peine que la mémoire utilisée par votre programme n'augmente indéfiniment. Le destructeur est à prévoir avec le contrôle d'accès public.

(4) Il faut maintenant prévoir les méthodes de votre classe. Commencez par écrire les sélecteurs de vos attributs privés et protégés : pour chacun d'eux, une méthode permettant de retourner la valeur de l'attribut et une méthode permettant de le modifier.
Pensez ensuite aux autres méthodes à inclure dans votre classe. Pensez simple : ne prévoyez pas des méthodes trop complexes.

Il y a une règle que vous pouvez mémoriser également pour savoir si vous n'avez pas trop surchargé votre classe : *ne prévoyez pas plus de 7 +/- 2 méthodes (sans compter les sélecteurs) dans vos classes*. Au-delà de 9, la compréhension de ces méthodes peut ne plus apparaître clairement pour l'utilisateur de votre classe. Et que faire si vous avez dépassé, de beaucoup, 9 méthodes ? Et bien, il faut fractionner votre classe en plusieurs classes, chacune d'elles ayant un sens et regroupant des attributs de la classe d'origine. Celle-ci se réduira de fait à l'inclusion d'attributs des sous-classes créées et vous réduirez ainsi le nombre de méthodes qui y figurent.

Chapitre 4

Les traitements

Dans ce chapitre nous allons nous focaliser sur ce que j'appelle les traitements, c'est-à-dire les fonctions au sens du langage C (non rattachées à une classe) et les méthodes (qui elles sont définies au sein d'une classe). Nous allons donc voir un ensemble de mécanismes valables aussi bien pour les fonctions que pour les méthodes. Ce chapitre vient donc en complément du précédent et une fois que vous en aurez terminé la lecture vous serez à même d'écrire des programmes en langage C++ dont la structure est simple. Il ne vous restera plus ensuite qu'à parcourir les autres chapitres à la découverte des mécanismes particuliers du langage C++.

4.1 PASSAGE D'ARGUMENTS PAR DÉFAUT

En langage C, il était indispensable que les fonctions soient appelées avec autant d'arguments que celles-ci en attendaient. Le langage C++ propose un mécanisme permettant d'assouplir cette règle. On peut préciser au compilateur qu'en cas d'absence d'un argument, celui-ci est remplacé par une valeur par défaut. L'attribution des valeurs aux paramètres étant réalisée par le rang[1], les paramètres par défaut doivent être les derniers arguments d'une fonction. Ce mécanisme consiste à fixer les valeurs par défaut dans la déclaration de la fonction, comme indiqué dans l'exemple suivant.

1. Cela signifie que lorsque vous appelez une fonction en lui passant des valeurs celles-ci sont attribuées dans l'ordre suivant : la première valeur est celle du premier argument, la seconde valeur est celle du second argument...

```
1  void f(int a, int b=2, int c=5); // prototype d'une fonction f avec deux arguments par défaut
2  ...
3  void main()
4  {
5   // appels possibles de la fonction f
6   f(10,6);    // a=10 et b=6 dans la fonction, c=5 par défaut
7   f(5);       // a=5 dans la fonction, b=2 et c=5 par défaut
8   f(6,4,1); // a=6, b=4, c=1 dans la fonction
9  }
```

Évidemment, ce mécanisme est également utilisable sur les méthodes des classes. Cette technique est souvent utilisée pour les constructeurs : on peut ainsi définir une collection de constructeurs ayant 0, 1 ou plusieurs arguments par défaut. Dans le cas où tous les arguments d'un constructeur possèdent une valeur par défaut, celui-ci peut faire office de constructeur par défaut (le constructeur sans argument) pour initialiser des objets. Cela vous offre donc, du point de vue de l'utilisation, de la flexibilité au détriment d'une complexification du code du constructeur en question. Celui-ci aura nécesairement un code plus compliqué à écrire.

4.2 ÉCHANGE D'OBJETS ENTRE FONCTIONS

Les objets peuvent être passés en arguments au même titre que les autres variables (par adresse, par valeur ou par référence). Il faut simplement noter qu'un objet d'une classe peut avoir accès aux membres privés d'un autre objet de la même classe, si une de ses opérations le reçoit en argument. En effet, la protection offerte par l'emploi de `public`, `private` ou `protected` s'effectue sur une classe, et non sur les objets. N'oubliez pas *la portée des contrôleurs d'accès est la classe et non l'objet*. Ainsi, le code ci-dessous compile parfaitement, malgré le référencement ligne 10 des attributs privés `pcIDEnom` et `pcIDEprenom` de l'objet `IDEP1`.

```
1  class Cidentite
2  {
3   ...
4   public :
5   int IDEcompare(Cidentite &);
6  };
7  ...
8  int Cidentite : :IDEcompare(Cidentite &IDEP1)
9  {
10  return( !strcmp(pcIDEnom,IDEP1.pcIDEnom) && !strcmp(pcIDEprenom,IDEP1.pcIDEprenom)) ;
11 }
```

Il ne faut pas oublier que :

(1) Dans le cas du passage par valeur, la fonction reçoit une copie de l'objet passé en argument. Elle peut faire ce qu'elle veut avec, cela n'a aucune incidence sur l'objet qui a été passé. Mais attention car dans ce cas il faudra se poser la question de la nécessité d'un constructeur par recopie (cf. annexe C puis section 3.2.3.).

(2) Dans le cas du passage par adresse, la fonction a accès à l'objet par son adresse. Toute modification s'effectue directement sur l'objet passé. On peut utiliser le qualificatif `const` afin d'interdire la modification de l'objet. L'accès aux membres utilise la notation "->" (cf. annexe C).

(3) Dans le cas du passage par référence, c'est effectivement l'adresse qui est transmise. Mais la gestion est laissée au compilateur et on continue d'utiliser la notation pointée pour référencer les membres. La modification est faite directement sur l'objet, mais on peut l'éviter en qualifiant l'argument avec `const` (cf. section 2.3.1.).

Comme pour les variables en langage C, il est possible de renvoyer en langage C++ un objet en retour d'une fonction. Celui-ci peut être retourné par valeur, par adresse ou par référence. Le cas de la transmission par valeur soulève encore le problème de la copie des pointeurs membres (on rencontre ici le même problème que celui soulevé dans la section 3.2.3.). Un second problème apparaît : lors de la transmission par adresse ou par référence, il ne faut pas renvoyer l'adresse d'un objet automatique créé dans la fonction[2]. *Ainsi, lorsqu'on retourne un objet automatique d'une fonction, on doit toujours le faire par valeur.*

4.3 PROPRIÉTÉS DES FONCTIONS MEMBRES

4.3.1. Spécification `inline`

Le compilateur traduit un appel de fonction par un saut en mémoire vers la zone de code correspondant à la fonction. L'avantage est que le code d'une fonction n'est présent qu'une seule fois en mémoire, et peut être appelé autant qu'il le faut. Le désavantage est que cela nécessite une gestion interne assez lourde, responsable d'une certaine lenteur d'exécution. La définition d'une fonction `inline` permet au compilateur de recopier le code de la fonction aux points d'appel au lieu de générer une instruction de saut. On économise ainsi un certain nombre de manipulations de mémoire et de pile. En contrepartie, la place mémoire occupée par le code augmente. Ce phénomène est illustré dans la figure 4.1 et se lit avec l'explication de la syntaxe à mettre en œuvre pour déclarer une méthode ou une fonction `inline`.

2. Certains compilateurs ne réinitialisent pas systématiquement en sortie de fonction la zone mémoire allouée aux objets automatiques. Dans ce cas, vous pouvez avoir l'impression qu'un objet automatique à une fonction est encore "en vie" lorsque vous sortez de la fonction : les valeurs des attributs sont toujours consultables... Mais il s'agit là d'un leurre : à la première occasion le compilateur réallouera cette zone mémoire et vous perdrez les valeurs des attributs des objets automatiques.

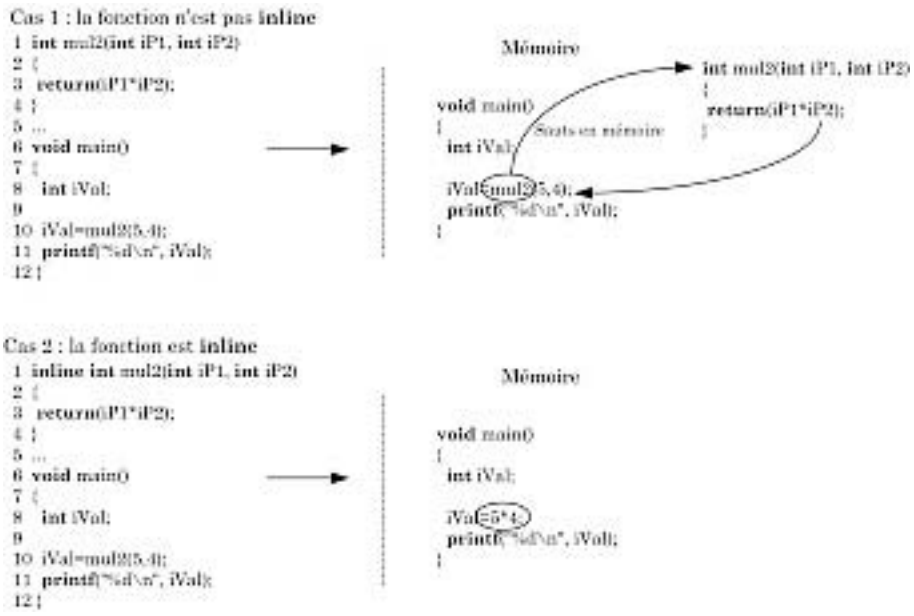


FIG. 4.1: Mécanisme des fonctions et méthodes `inline`

```
1  inline int mul2(int iP1, int iP2)
2  {
3   return(iP1*iP2);
4  }
5  ...
6  void main()
7  {
8   int iVal;
9
10  iVal=mul2(5,4);
11  printf("%d\n", iVal);
12 }
```

La spécification de fonction `inline` est possible pour les membres d'une classe. Vous avez deux façons de faire. Une méthode est `inline` si l'on place sa définition dans la déclaration de la classe (fichier `.h`). Dans l'exemple ci-dessous, l'accesseur `IDElire_nom` est déclaré comme étant une méthode `inline`.

```
1  class Cidentite
2  {
3   ...
4   public :
```

```
5    char *IDElire_nom()
6    {
7      return (char *)pcIDEnom;
8    }
9  };
```

Il existe une seconde façon de déclarer une méthode `inline`, beaucoup plus explicite puisqu'elle requiert l'utilisation du mot-clef `inline`. Pour cela, il suffit de faire précéder l'interface de la méthode, lors de sa définition, par le mot-clef `inline` comme dans le cas d'une fonction `inline`. Néanmoins, vous ne pouvez plus définir cette méthode dans le corps de la classe, c'est-à-dire dans le fichier `.cpp`, mais vous devez le faire dans le fichier contenant sa déclaration (le fichier `.h`). Voici le contenu du fichier `Cidentite.h` dans le cas de la classe `Cidentite`. Cela provoque le même résultat que dans l'exemple ci-dessus.

```
1  class Cidentite
2  {
3   ...
4   public :
5    char *IDElire_nom();
6   ...
7  };
8
9  inline char * Cidentite : :IDElire_nom()
10 {
11  return (char *)pcIDEnom;
12 }
```

Et que se passe-t-il si jamais vous vous trompez, c'est-à-dire si vous mettez quand même la définition de la méthode `inline` dans le fichier `.cpp` ? Et bien vous n'aurez aucune erreur de compilation ou de linkage dans les fichiers de la classe, mais par contre vous aurez un message d'erreur au linkage à chaque ligne où vous appelez cette méthode : *symbole externe non résolu*. Le corps de la méthode `inline` n'a pas été trouvé par le linker.

Quand et comment déclarer des fonctions/méthodes `inline` ? Tout d'abord vous ne déclarez des fonctions ou méthodes `inline` que lorsque celles-ci ne contiennent pas beaucoup de lignes de code car sinon vous aurez un programme trop volumineux au final (le fichier `.exe` généré pourra avoir une taille trop grande). *Typiquement, on déclare tous les accesseurs situés dans des classes comme des méthodes `inline`.*
Justement, dans le cas de méthodes `inline`, quelle syntaxe utiliser ? Et bien,

d'un point de vue génie logiciel je vous conseille de ne pas alourdir la lecture de l'interface de votre classe. En pratique cela signifie que la déclaration des accesseurs `inline` peut se faire directement lors de la déclaration de la classe (première syntaxe vue dans cette section), tandis que les autres méthodes que vous voudrez mettre `inline` (plus volumineuses sans doute) le seront à l'aide de la seconde syntaxe.

4.3.2. Méthodes statiques

Dans la section 3.3 nous avons vu qu'il était possible de déclarer des variables et des attributs d'une classe avec le mot-clef `static`. Cela avait pour effet "d'allonger la durée de vie" de la variable ou de l'attribut en question.

En C++ il est possible de déclarer qu'une méthode d'une classe est `static`. Une méthode non statique ne peut être appelée que via un objet (en utilisant la notation pointée ou la notation fléchée) et dans cette méthode on a accès aux membres de l'objet. Si la méthode est statique, elle peut être appelée soit via un objet, soit par spécification de la classe d'appartenance avec l'opérateur de résolution de portée. Pour déclarer qu'une méthode est `static`, il suffit de faire précéder son interface du mot-clef `static`.

```
1  class Cidentite
2  {
3   ...
4   public :
5     static int IDEcompter() ;
6  };
7  ...
8  void main()
9  {
10  Cidentite IDEclient1 ;
11  ...
12  IDEclient1.IDEcompter() ; // Appel correct
13  Cidentite : :IDEcompter() ; // Appel correct
14  ...
15 }
```

Il est important de noter qu'il y a quelques précautions à prendre concernant la déclaration de telles méthodes. En effet, une méthode `static` pouvant être appelée indépendemment de tout objet elle ne doit absolument pas référencer des attributs... qui ne seraient pas eux-mêmes déclarés en `static` !

Pour s'en convaincre, il suffit de reprendre la ligne 13 de l'exemple ci-dessus et de supposer que la méthode `IDEcompter` référence dans son code l'attribut `pcIDEnom`. Vous avez là une incohérence puisque le compilateur ne sait pas, ligne

13, quel est l'objet sur lequel il doit travailler. Si vous faites cette erreur vous aurez un message d'erreur à la compilation.

Comme cela a été souligné précédemment, vous pouvez référencer dans des méthodes `static` des attributs `static` puisque ceux-ci ont une durée de vie qui est liée au programme et non pas aux objets.

À ce stade vous devez vous poser la question de savoir *quand est-ce qu'on déclare une méthode `static`...* et bien, d'un point de vue général, vous allez le faire dès lors que vous avez besoin de réaliser un traitement sur tous les objets d'une même classe et ce sans passer par des variables globales. Un exemple concret ? Compter le nombre d'objets d'une classe : vous déclarez un attribut `static`, vous incrémentez cet attribut dans les constructeurs et vous prévoyez un accesseur en lecture qui soit une méthode `static`.

4.3.3. Auto-référence d'une classe

Chaque classe possède un pointeur "caché", noté `this`. Lors de la création d'un objet, ce pointeur désigne l'emplacement mémoire où l'objet est stocké. Par conséquent, dans la déclaration d'une méthode référencer un attribut, par exemple, `pcIDEnom` ou `this->pcIDEnom`[3] revient au même... d'un point de vue langage C++. *En effet, d'un point de vue génie logiciel on évitera à tout prix la seconde notation qui alourdie inutilement la lecture de vos programmes*. Ce pointeur est transmis aux fonctions membres de manière transparente à l'utilisateur, par la pile[4].

Là encore, dans quel cas utiliser le pointeur `this` ? Et bien, dans certaines situations bien particulières... C'est le cas lors de la création d'une classe représentant des éléments d'une liste chaînée : l'opération d'ajout dans la chaîne doit nécessairement connaître l'adresse de l'objet en cours pour pouvoir le placer à la suite de son prédécesseur.

4.3.4. Fonctions membres constantes

Nous avons déjà vu qu'il était possible de déclarer qu'un attribut ou un objet était constant, c'est-à-dire non modifiable. Pour cela il suffit de faire précéder sa déclaration du mot-clef `const`. En langage C++ il est également possible de déclarer qu'une méthode est constante. Quel est l'intérêt ? Et bien supposez que vous ayez déclaré un objet constant, c'est-à-dire un objet dont le contenu ne doit pas être modifié... puisqu'il est constant. Or, vous avez sans aucun doute écrit dans la classe qui lui est associé des méthodes qui modifient des attributs. Cela veut dire que vous pouvez, en appelant ces méthodes sur votre objet constant, modifier son contenu... ce qui est

3. Notez bien ici l'utilisation de la notation fléchée, puisque `this` est un pointeur.

4. Le premier argument lors de l'appel d'une fonction est toujours le pointeur `this`. Cela implique beaucoup de choses, notamment que le premier travail du compilateur C++ est de transformer les methodes en fonctions en y ajoutant le paramètre `this`... ce qui revient à transformer le code C++ en code C !

contradictoire avec la définition même d'un objet constant. Pour cette raison, en langage C++, on ne peut appeler sur des objets déclarés constants que des méthodes qui ont elles-mêmes été déclarées constantes. Si vous voulez, c'est un peu comme donner le droit ou pas à une méthode de s'exécuter sur un objet constant ou pas. D'un point de vue syntaxique, il suffit de faire suivre l'interface de la méthode du mot-clef `const`.

```
1  class Cidentite
2  {
3   ...
4   public :
5     char *IDElire_nom() const ;
6     char *IDElire_prenom() ;
7  };
8  ...
9  void main()
10 {
11  const Cidentite IDEclient1 ;
12  ...
13  IDEclient1.IDElire_nom() ; // Appel correct
14  IDEclient1.IDElire_prenom() ; // Appel incorrect
15  ...
16 }
```

Dans l'exemple ci-dessus, la ligne 11 déclare un objet constant nommé `IDEclient1` et initialisé par appel au constructeur par défaut. Sur la ligne 13, la méthode `IDElire_nom` est appelée sur cet objet. L'appel est correct puisque cette méthode est constante, ce qui n'est pas le cas de la méthode `IDElire_prenom` : la ligne 14 provoquera une erreur de compilation.

Notez bien ici le paradoxe, sur un plan purement conceptuel : en théorie rien ne vous empêche d'écrire au sein de votre classe une méthode constante qui modifie tout ou partie de ses attributs. Évidemment vous auriez des erreurs de compilation, mais sur un plan conceptuel cela fait plutôt désordre d'arriver à un tel résultat. Au bout du compte, comme souvent, il est plus de la responsabilité du concepteur de choisir les méthodes qui doivent être constantes en leur gardant leur sens premier : celui de ne pas modifier le contenu des objets.

Il est possible de prévoir des méthodes en fonction de la présence du mot-clef `const`. Ainsi on peut avoir une version de la méthode pour le cas où l'objet ne serait pas constant, et une version pour le cas contraire. L'appel à la bonne méthode se fait automatiquement en fonction de l'objet concerné.

```
1  class Cidentite
2  {
3   ...
```

```
4   public :
5     char *IDElire_nom() const;
6     char *IDElire_nom();
7   };
8   ...
9   void main()
10  {
11    const Cidentite IDEclient1;
12    Cidentite IDEclient2;
13    ...
14    IDEclient1.IDElire_nom(); // Appel de la version constante
15    IDEclient2.IDElire_nom(); // Appel de la version non constante
16    ...
17  }
```

4.3.5. Pointeurs sur les membres d'une classe

Si vous êtes un pro de la programmation en langage C, vous devez alors connaître les pointeurs de fonctions. De quoi s'agit-il ? Et bien tout simplement d'un pointeur non pas sur un objet mais sur une fonction. On peut de cette manière appeler différentes fonctions ayant la même en-tête avec un seul et même pointeur... correctement affecté. L'exemple ci-dessous vous montre comment procéder en langage C (cela reste d'actualité en langage C++).

```
1   void f1(int, float); // Une premiere fonction
2   void f2(int, float); // Une seconde fonction
3   void (*f)(int, float); // Un pointeur compatible avec les 2 fonctions précédentes
4
5   void main()
6   {
7     ...
8     f = f1;
9     (*f)(10,42.5); // Appel de f1 via le pointeur f
10    f = f2;
11    (*f)(10,42.5); // Appel de f2 via le pointeur f
12  }
```

Dans cet exemple, vous constatez sur la ligne 3 la déclaration d'un pointeur de fonction, ce pointeur prenant en argument un premier paramètre de type `int` et un second de type `float`. Cela implique que ce pointeur ne peut pointer que sur des fonctions qui prennent les mêmes arguments (c'est le cas des fonctions `f1` et `f2`).

En langage C++ il est possible de réaliser des pointeurs sur des méthodes d'une classe. On prend en compte dans la signature du pointeur sur les méthodes, la

classe d'appartenance de la fonction. La syntaxe suit le même esprit que dans le cas des pointeurs de fonctions et leur utilisation est semblable à celle en langage C. L'exemple suivant illustre par la pratique comment déclarer et utiliser des pointeurs sur des méthodes.

```
1  void (Cidentite : :*IDEaffecter)(const char *pcP) ; // Un pointeur de méthode
2
3  void main()
4  {
5   ...
6   Cidentite IDEclient1 ;
7   ...
8   IDEaffecter = Cidentite : :IDEaffecter_nom ;
9   (IDEclient1.*IDEaffecter)("Durant");
10 }
```

Notez bien au passage que cela n'est possible que sur des méthodes qui sont déclarées `public`, puisque sinon la ligne 8 ci-dessus ne compile pas (vous n'avez pas accès à la méthode `IDEaffecter_nom`).

Chapitre **5**

Les fonctions et les classes amies

5.1 AMIS ET FAUX AMIS : LE POINT DE VUE DU GÉNIE LOGICIEL

La notion d'amitié a été introduite en langage C++ et est fortement liée à l'encapsulation des données, c'est-à-dire au fait que les attributs peuvent être inaccessibles de l'extérieur de la classe en utilisant les contrôles d'accès `private` ou `protected`. Ainsi, comme nous l'avons vu dans la section 3.2 le code suivant provoque une erreur de compilation sur la ligne 10.

```
1  class Cidentite
2  {
3   private :
4     char * pcIDEnom ;
5    ...
6  };
7  ...
8  void Afficher_nom(Cidentite IDEP1)
9  {
10  printf("%s\n",IDEP1.pcIDEnom);
11 }
```

Il est néanmoins, techniquement, possible de faire en sorte que ce code compile, c'est-à-dire que la fonction `Afficher_nom` puisse accéder à l'attribut privé

pcIDEnom sans pour autant faire partie de la classe Cidentite. Pour cela, il suffit de déclarer que cette fonction est amie de la classe. Et c'est justement là qu'est le problème d'un point de vue génie logiciel puisqu'en faisant cela on viole l'encapsulation de l'attribut pcIDEnom telle qu'elle avait été décidée par le concepteur de la classe. Vous pouvez voir le mécanisme d'amitié comme un mécanisme permettant de rendre public à certaines fonctions des membres privés.

> L'utilisation de l'amitié dans une démarche de génie logiciel est fortement déconseillée. Si vous souhaitez accéder dans une fonction à des attributs privés, il vous suffit d'utiliser les accesseurs présents dans les classes.

Dans la suite de ce chapitre nous allons voir quelles sont les situations où l'on peut déclarer des amitiés en langage C++.

5.2 LE CAS D'UNE FONCTION AMIE D'UNE CLASSE

Ce cas de figure est celui de l'exemple précédent. Pour rendre une fonction amie d'une classe, et donc lui permettre d'accéder à tous ses éléments privés et protégés, il suffit d'inclure son interface précédée du mot-clef friend au sein de la déclaration de la classe. Dans l'exemple précédent cela nous donne le code ci-dessous (regardez la ligne 12). La fonction peut être définie dans un fichier .cpp différent du fichier Cidentite à partir du moment où le fichier .h de la classe y est inclus.

```
1  class Cidentite
2  {
3   private :
4     char pcIDEnom[100];
5     char pcIDEprenom[100];
6
7   public :
8     void IDEaffecter_nom(const char *pcn);
9     void IDEaffecter_prenom(const char *pcn);
10    void IDEafficher();
11
12    friend void Afficher_nom(Cidentite IDEP1);
13 };
```

5.3 LE CAS D'UNE MÉTHODE D'UNE CLASSE AMIE D'UNE AUTRE CLASSE

Pour déclarer qu'une méthode est amie d'une classe on utilise la même syntaxe que dans le cas d'une fonction sauf que l'on utilise son nom long pour la nommer dans la classe (si vous ne faites pas ça on se retrouve dans le cas de la déclaration d'amitié d'une fonction au sens du langage C). Reprenons l'exemple de la fonction `Afficher_nom` et supposons maintenant qu'il s'agisse d'une méthode de la classe `Cpersonne`.

```
1  #include "Cpersonne.h"
2
3  class Cidentite
4  {
5   private :
6    char pcIDEnom[100];
7    char pcIDEprenom[100];
8
9   public :
10   void IDEaffecter_nom(const char *pcn);
11   void IDEaffecter_prenom(const char *pcn);
12   void IDEafficher();
13
14   friend void Cpersonne : :Afficher_nom(Cidentite IDEP1);
15 };
```

Notez bien que pour que la ligne 14 compile il est nécessaire que la classe `Cpersonne` ait été déclarée au préalable, d'où l'inclusion de la ligne 1.

5.4 TOUTES LES FONCTIONS MEMBRES D'UNE CLASSE AMIES D'UNE AUTRE CLASSE

La notion d'amitié porte également sur les classes, c'est-à-dire que vous pouvez déclarer qu'une classe est amie d'une autre. Cela implique que toutes les méthodes de la classe amie ont accès aux membres, sans restriction, de cette autre classe. La syntaxe de déclaration d'amitié reste simple et semblable à ce que nous avons déjà vu dans ce chapitre : il suffit dans l'interface de la classe donnant son amitié d'inclure une ligne spécifiant la relation d'amitié.

`friend class` nom_classe ;

Voici un exemple où nous supposons que la classe `Cpersonne` à laquelle nous avons fait référence dans la section précédente soit amie de la classe `Cidentite`.

Cela veut donc dire que toutes les méthodes de la classe `Cpersonne` peuvent accèder à tous les membres (qu'ils soient privés, protégés ou publics) de la classe `Cidentite`.

```
1  #include "Cpersonne.h"
2
3  class Cidentite
4  {
5   private :
6     char pcIDEnom[100];
7     char pcIDEprenom[100];
8
9   public :
10    void IDEaffecter_nom(const char *pcn);
11    void IDEaffecter_prenom(const char *pcn);
12    void IDEafficher();
13
14    friend class Cpersonne;
15 };
```

Notez bien que l'amitié entre classe n'est pas transitive c'est-à-dire que si, dans l'exemple ci-dessus, il existait une classe `Cfoule` amie de la classe `Cpersonne` nous n'aurions pas de relation d'amitié entre `Cfoule` et `Cidentite`. Donc, la classe `Cfoule` n'aurait accès qu'aux membres publics de la classe `Cidentite`.

Chapitre 6

Les exceptions

6.1 GESTION DES EXCEPTIONS EN LANGAGE C++

Nous avons déjà vu dans la section 1.3.2. le mécanisme des exceptions d'un point de vue général. Relisez cette section si tout n'est pas clair dans votre esprit car dans ce chapitre nous allons principalement voir comment ce mécanisme s'applique en langage C++.

Tout d'abord, une première différence avec la présentation faite dans la section 1.3.2. est lié à l'élément remonté lors d'une levée d'exception. Dans la section 1.3.2. l'élément remonté était un état ("Allocation_impossible"...) tandis qu'en langage C++ l'élément remonté est une donnée (une variable, un objet d'une classe...). Nous allons donc nous organiser en conséquence pour structurer notre développement. Ainsi nous allons créer une classe spéciale, nommée `Cexception`, et qui permettra de créer les objets levés lors d'une exception. Cette classe contient, dans sa version de base (cf. annexe E), un attribut unique nommé `uiEXCvaleur` et qui contient la valeur de l'exception levée. Son état si vous préférez. Libre à vous par la suite, au fil de vos développements, de raffiner cette version basique en y ajoutant d'autres attributs qui contiendront les informations que vous voulez faire remonter lors d'une levée d'exception.

Dans la section 1.3.2. nous avons vu que la gestion théorique des exceptions nécessitait l'utilisation de deux fonctions **LeverException** et **EnCasException** pour lever et attraper une exception. Nous allons voir dans les sections qui suivent ce que ces fonctions deviennent en langage C++.

6.1.1. Lever une exception

Pour lever une exception en langage C++ il faut utiliser l'opérateur `throw` suivit de l'objet à faire remonter. Dans l'exemple suivant nous supposons l'existence d'une classe `Cfraction` pour la gestion des fractions du type $\frac{nominateur}{denominateur}$ où $denominateur$ est forcément non nul. Voici la définition d'un constructeur à deux arguments pour lequel nous faisons également figurer ses spécifications pour faire apparaître que la gestion en postcondition par exception implique qu'il n'y a pas de précondition sur le dénominateur. Notez que dans cet exemple apparaissent un extrait du fichier `Cfraction.h` (partie déclaration de la classe) et un extrait du fichier `Cfraction.cpp` (définition du constructeur).

```
1  #include "Cexception.h"
2
3  #define denominateur_nul 100
4
5  class Cfraction
6  {
7   private :
8     unsigned int uiNominateur ;
9     unsigned int uiDenominateur ;
10
11  public :
12    Cfraction(unsigned int uiNom, unsigned int uiDen) ;
13    ...
14 };
15 ...
16 /*****************************************************************
17 Nom : Cfraction
18 *****************************************************************
19 Constructeur à deux arguments permettant d'initialiser une fraction
20 *****************************************************************
21 Entrée : le nominateur et le dénominateur
22 Nécessite : néant
23 Sortie : rien
24 Entraîne : (L'objet en cours est initialisé) ou
25           (Exception denominateur_nul : Le dénominateur est nul)
26 *****************************************************************/
27 Cfraction : :Cfraction(unsigned int uiNom, unsigned int uiDen)
28   {
29    if (uiDen==0)
30    { // Situation d'exception : nous cherchons à créer une fraction inexistante
31          Cexception EXCobjet ;
32          EXCobjet.EXCmodifier_valeur(denominateur_nul) ;
33          throw(EXCobjet) ;
34    }
```

```
35    // La fraction est initialisable
36    uiDenominateur=uiDen ;
37    uiNominateur=uiNom ;
38  }
```

Vous remarquerez dans cet exemple plusieurs points importants lors de la mise en place d'une levée d'exceptions :

(1) *Informer la classe utilisatrice*. Considérez la partie déclarative de la classe (fichier `Cfraction.h`) : il faut tout d'abord *inclure l'interface de la classe* `Cexception` pour pouvoir ensuite créer des objets qui seront levés lors d'une exception (ligne 1). Mais le point le plus important réside dans la ligne 3 : en *attribuant une valeur numérique à un nom d'exception* nous sommes en train de déclarer une catégorie d'exceptions. Ainsi la classe utilisatrice (elle incluera l'interface `Cfraction.h` dans cet exemple) va pouvoir recevoir des objets de la classe `Cexception` ayant cette valeur. Enfin, dans le corps de la classe on retrouve, lignes 24 et 25, *la déclaration en postcondition de la levée d'exception* qui peut être réalisée dans le constructeur. Remarquez que cette déclaration doit également apparaître sur la ligne 13 pour être tout à fait complète.
Au final, tout programmeur qui va utiliser la classe `Cfraction` est donc pleinement informé de quelles fonctions peuvent lever quelles exceptions.

(2) *Gérer l'exception dans les fonctions qui la lèvent*. Chaque méthode ou fonction qui lève une exception doit intégrer une partie de code spécifique. Dans l'exemple de la classe `Cfraction`, il s'agit des lignes 29 à 34. Remarquez qu'un test (ligne 29) permet de détecter si la méthode est dans une situation d'exception ou pas. Si tel est le cas, les lignes 30 à 34 sont exécutées. Elles provoquent la création d'un objet de type `Cexception` (initialisé par appel à son constructeur par défaut) puis l'affectation de la valeur `denominateur_nul` à cet objet. L'exception est ensuite levée à l'aide de la commande `throw` (ligne 33).

6.1.2. Attraper une exception

La question qui se pose ici est la suivante : quand j'appelle une fonction/méthode levant une exception comment faire pour déclencher un traitement particulier ?
La réponse est simple : il suffit d'utiliser les opérateurs `try` et `catch`. Tout d'abord il faut créer un bloc `try` qui contient les instructions pouvant lever une exception. À la suite de ce bloc, il faut faire figurer un ou plusieurs *gestionnaires d'exception* également appelés blocs `catch`. Reprenons l'exemple de la classe `Cfraction` introduit dans la section précédente et supposons l'existence de la fonction `main` suivante.

```
1  #include "Cfraction.h"
2  #include <stdio.h>
3
4  void main()
```

```
5  {
6          try { // Bloc try : on y inclut les lignes pouvant lever une exception
7                  Cfraction FRCV1(5,0);
8          }
9          catch(Cexception EXClevee)
10         { // Bloc catch : on spécifie le traitement à réaliser
11          printf("L'exception n°%d a été levée \n",EXClevee.lire_valeur());
12         }
13         printf("J'exécute la suite de la fonction main\n");
14 }
```

Remarquez la ligne 1 : l'inclusion de l'interface de la classe `Cfraction` provoque l'inclusion de l'interface de la classe `Cexception` puisque celle-ci est incluse dans `Cfraction.h`. Cela permet, ligne 9, de créer un objet de la classe `Cexception` sans rencontrer de message d'erreur à la compilation.
Passons maintenant à l'analyse du code ci-dessus. La ligne pouvant lever une exception est la ligne 7 puisqu'on fait appel au constructeur à deux arguments de la classe `Cfraction` et qu'il est écrit (cf. section précédente) qu'il peut lever l'exception `denominateur_nul`. Cette ligne est donc incluse dans un bloc `try`. Étant donnés les paramètres passés au constructeur, l'exception va être levée et un objet de la classe `Cexception` va être remonté sur la ligne 7. Le compilateur va alors exécuter le code du gestionnaire d'exceptions qui prend en paramètre un objet de cette classe : autrement dit, les lignes 10 à 12 vont être exécutées avant de passer à la ligne 13. Notez bien que si l'exécution de la ligne 7 ne provoquait pas de levée d'exception, nous passerions directement de la ligne 7 à la ligne 13.

Maintenant considérons une version légèrement modifiée de l'exemple précédent (l'instruction `printf` de la ligne 13 est passée dans le bloc `try`).

```
1  #include "Cfraction.h"
2  #include <stdio.h>
3
4  void main()
5  {
6          try { // Bloc try : on y inclut les lignes pouvant lever une exception
7                  Cfraction FRCV1(5,0);
8                  printf("J'exécute la suite de la fonction main\n");
9          }
10         catch(Cexception EXClevee)
11         { // Bloc catch : on spécifie le traitement à réaliser
12          printf("L'exception n°%d a été levée \n",EXClevee.lire_valeur());
13         }
14 }
```

Au niveau de l'affichage écran que se passe-t-il ? Et bien contrairement à la première version, la phrase **J'exécute la suite de la fonction main** n'est pas affichée à l'écran. Pourquoi ? Et bien tout simplement car la ligne 7 levant une exception, il va y avoir un saut directement à la ligne 12 sans exécuter la ligne 8. Cet exemple soulève un point crucial dans la délimitation des blocs `try` : *Quelles lignes de code doit-on mettre dans un bloc `try` ?* Par exemple, nous pourrions mettre tout le code d'une fonction dans un unique bloc `try` avec l'ensemble des blocs `catch` à la fin. Mais dans ce cas, la levée d'une exception provoque la non-exécution de la totalité de la fonction. À l'inverse, si vous mettez un bloc `try` par instruction levant une exception (suivi des blocs `catch` qui lui sont associés) vous risquez de diminuer la lisibilité de votre code. Malheureusement, il n'y a pas de règle pour bien définir le contenu de vos blocs `try` : cela dépend uniquement de ce que vous voulez faire faire au programme.

Voici en synthèse quelques règles relatives à la récupération d'exceptions levées :

(1) Un bloc `try` est nécessairement suivi d'un ou plusieurs bloc `catch`.

(2) Aucune ligne de code ne peut s'intercaler entre un bloc `try` et un bloc `catch` ou entre deux blocs `catch` consécutifs.

(3) Lors de la récupération d'une exception, est exécuté le premier gestionnaire d'exceptions (dans l'ordre d'écriture du programme) dont l'argument correspond au type de l'objet levé.

6.1.3. Quel gestionnaire d'exceptions choisir ?

Nous nous intéressons dans cette section à la mise en place de gestionnaires d'exceptions et notamment à la façon dont le compilateur choisit le gestionnaire à exécuter.

Que se passe-t-il lorsque plusieurs blocs `catch` suivent un bloc `try` ? Autrement dit, quel est le gestionnaire d'exceptions qui va être choisi ? Ce choix s'effectue en fonction du type de l'objet levé, c'est-à-dire le type de l'objet passé à l'opérateur `throw`. Lorsqu'on sort d'un bloc `try` avec une exception, les cas suivants sont examinés, **par ordre d'apparition dans le code** selon l'algorithme suivant (on commence par regarder le premier bloc `catch` qui suit le bloc `try`) :

(1) Si le bloc `catch` possède un argument exactement du même type que celui de l'objet passé par `throw` alors ce gestionnaire est choisi.

(2) Sinon si le bloc `catch` possède un argument d'un type correspondant à une classe de base[1] de l'objet passé par `throw` alors ce gestionnaire est choisi.

1. La notion de classe de base vous est présentée dans le chapitre 7 sur l'héritage.

(3) Sinon si le bloc `catch` possède un argument d'un type correspondant à un pointeur sur une classe dérivée[2] de l'objet passé par `throw` alors ce gestionnaire est choisi.

(4) Sinon si le bloc `catch` possède un argument d'un type indéterminé (noté `...`) alors ce gestionnaire est choisi.

(5) Sinon, considérer le bloc `catch` suivant (s'il n'en existe pas alors l'exception est relayée à la fonction appelante).

La recherche s'arrête dès qu'un gestionnaire correspond. La sélection se fait suivant l'ordre de lecture. Ainsi, si un gestionnaire du type `catch(...)` est défini en premier, il interceptera tous les appels. Ce type de gestionnaire est utilisé lorsque l'on désire offrir un traitement pour une exception que l'on n'avait pas prévue (mais dans ce cas on le mettra comme dernier gestionnaire). Si aucun gestionnaire ne peut être trouvé alors la fonction en cours est interrompue et l'exception est transmise à la fonction appelante.

Considérez l'exemple ci-dessous, fictif, et pour lequel nous ne savons pas encore le type de l'objet levé par `throw` dans la fonction `g` (la définition de cette dernière fonction ne figure pas dans l'exemple).

```
1  #include <stdio.h>
2  #include "Cexception.h"
3
4  void f()
5  {
6          try {
7                  g();
8          }
9          catch(Cexception EXCP1)
10         {
11                 printf("Gestionnaire n°1\n");
12         }
13         catch(int iP1)
14         {
15                 printf("Gestionnaire n°2\n");
16         }
17         catch(...)
18         {
19                 printf("Gestionnaire n°3\n");
20         }
21 }
22
23 void main()
24 {
```

2. La notion de classe dérivée vous est présentée dans le chapitre 7 sur l'héritage.

```
25  f();
26 }
```

Examinons trois cas de figure différents :

(1) ***La fonction `g` lève, par l'opérateur `throw`, un objet de type `Cexception`.*** Dans ce cas l'application de l'algorithme de choix précédent va conduire le compilateur à choisir le gestionnaire n°1 puisqu'il est le premier examiné et qu'il possède un argument du type `Cexception`.

(2) ***La fonction `g` lève, par l'opérateur `throw`, un objet de type `int`.*** Dans ce cas l'application de l'algorithme de choix précédent va conduire le compilateur à choisir le gestionnaire n°2. Dans un premier temps, le gestionnaire n°1 va être examiné mais le type de son argument est `Cexception` pas `int`. On passe donc à l'examen du second gestionnaire[3] qui est retenu puisqu'il possède un argument du type `int`.

(3) ***La fonction `g` lève, par l'opérateur `throw`, un objet d'un type autre que `int` et `Cexception`.***[4] Dans ce cas l'application de l'algorithme de choix précédent va conduire le compilateur à choisir le gestionnaire n°3. Dans un premier temps, les gestionnaires n°1 et 2 vont être examinés et ne seront pas retenus. Or le troisième gestionnaire possède le type "indéterminé" ce qui veut dire qu'il sera choisi quel que soit le type de l'objet renvoyé par le `throw` réalisé dans la fonction `g`.

Notez bien que si le gestionnaire n°3 avait été mis sur la ligne 9 juste après le bloc `try` alors dans les trois cas de figure ci-dessus, c'est toujours le gestionnaire n°3 qui aurait été choisi (car il possède l'argument "indéterminé"). C'est pour cette raison qu'il est toujours mis à la fin.

Enfin, notez que si le gestionnaire n°3 n'était pas présent, la levée d'un objet d'un type autre que `int` et `Cexception` provoquerait la remontée de cet objet dans la fonction qui appelle la fonction `f`, c'est-à-dire la fonction `main`. Or, aucuns blocs `try` et `catch` n'y ont été intégrés : la fonction `main` ne sait donc pas gérer les exceptions levées et comme il s'agit de la première fonction du programme celui-ci sera interrompu. Vous aurez alors à l'écran l'apparition d'un message d'erreur lors de l'exécution du programme.

6.1.4. Gestion hiérarchisée des exceptions

Contrairement à la vision générale du mécanisme des exceptions donnée dans le chapitre 1 et aux exemples présentés jusqu'ici dans ce chapitre, la levée d'exceptions ne porte pas que sur les fonctions, mais plutôt sur les blocs. Autrement dit, *l'exécution*

3. Je précise quand même que dans certains cas particuliers ce gestionnaire pourrait être choisi si le compilateur savait comment convertir un `int` en `Cexception`. Les conversions sont étudiées dans le chapitre 8 sur la surcharge.

4. Notez également que dans ce cas de figure, je suppose que le type de l'objet levé n'est pas convertible en un objet de type `int` (ce serait le cas d'un `float` notamment) ou de type `Cexception`.

d'un `throw` *ne provoque pas la sortie de la fonction en cours, mais la sortie du bloc en cours.* Pour illustrer cela, voici une portion de code et l'affichage correspondant à l'écran.

```
1  #include <stdio.h>
2
3  void f()
4  {
5   int iBoucle;
6   try {
7                   printf("Début d'exécution de la fonction f\n");
8                   for (iBoucle=0;iBoucle<5;iBoucle ++)
9                           if (iBoucle==3)
10                          {
11                           throw iBoucle;
12                          }
13          }
14   catch(int iP1)
15          {
16                  printf("Gestionnaire n°3\n");
17          }
18   catch(...)
19          {
20                  printf("Gestionnaire n°4\n");
21          }
22 }
23
24 void main()
25 {
26   try {
27                  f();
28          }
29   catch(int iP1)
30          {
31                  printf("Gestionnaire n°1\n");
32          }
33   catch(...)
34          {
35                  printf("Gestionnaire n°2\n");
36          }
37 }
```

Il est intéressant de noter que la sortie à l'écran est la suivante :

```
Début d'exécution de la fonction f
Gestionnaire n°3
```

Cela montre bien que, malgré l'exécution de l'opérateur `throw` dans la fonction `f` le compilateur n'examine pas les gestionnaires d'exceptions présents dans la fonction `main`. Ici, l'exception levée est gérée par la fonction qui la lève ce qui est différent du mécanisme général des exceptions tel que nous l'avons introduit. Le niveau de remontée lors de la levée d'une exception est le bloc et non pas la fonction.

Il est possible, dans un bloc `catch`, de réemettre une exception attrapée. Pour cela, il suffit dans le bloc `catch` de simplement écrire :

```
throw;
```

6.1.5. Liste d'exceptions valides

On peut associer à chaque fonction une liste d'exceptions valides. Si une exception n'appartenant pas à cette liste est levée dans la fonction, une erreur d'exécution se produit et la fonction `unexpected` est appelée. Celle-ci, par défaut, met fin au programme en appelant la fonction `terminate`.
Pour déclarer une liste d'exceptions pouvant être levées dans une fonction, il suffit à la suite de son interface (**aussi bien dans la définition que dans la déclaration de la fonction**) de préciser les types des objets pouvant être levés.

```
type nom_fonction(parametres) throw(type1,type2,...)
```

L'exemple suivant montre un cas, dont l'exécution provoque le même résultat que si nous n'avions pas précisé les types d'exceptions valides.

```
1  void f() throw(Cexception,int)
2  {
3   int iBoucle;
4   printf("Début d'exécution de la fonction f\n");
5   for (iBoucle=0;iBoucle<5;iBoucle ++)
6           if (iBoucle==3)
7                           {
8                            throw iBoucle;
9                           }
10 }
```

Supposons maintenant que, ligne 8, nous levions non plus un `int` mais un `char`, par exemple en écrivant :

```
throw "c";
```

Lors de l'exécution de la fonction `f`, et en cas de levée de l'exception, vous aurez alors un message d'erreur puisqu'une exception non autorisée a été levée.
Attention, cette spécification d'exception, qui même si elle est définie dans la norme ANSI du langage C++, n'est pas forcément reconnue par tous les compilateurs (c'est notamment le cas de l'environnement Microsoft Visual C++ 6.0).

En résumé, la spécification d'exceptions peut prendre les formes suivantes :

Syntaxe	Sens
`...f(...)`	Toutes les exceptions peuvent être levées dans la fonction `f`.
`...f(...) throw(type1,type2,...)`	Seules les exceptions du type `type1`, `type2`,..., peuvent être levées dans la fonction `f`.
`...f(...) throw()`	Aucune exception ne peut être levée dans la fonction `f`.

6.2 FONCTIONS LIÉES AU TRAITEMENT DES EXCEPTIONS

Le langage C++ offre différentes fonctions associées à la gestion des exceptions. Imaginez qu'une exception levée ne soit attrapée par aucune des fonctions du programme. Le programme est donc interrompu, comme nous l'avons déjà souligné à de multiples reprises. Le point important est que cet arrêt n'est pas le même que dans le cas d'un arrêt "normal". En langage C++ la levée de certaines exceptions peut être remplacée pour l'appel à une fonction définie par l'utilisateur, tandis que d'autres ne sont pas modifiables. Cela veut dire que l'on peut contourner, d'une certaine manière l'arrêt du programme.
Nous allons dans un premier temps voir quelles sont les différentes façons de terminer un programme, avant d'expliquer comment remplacer l'appel à certaines fonctions de terminaison par vos propres fonctions.

6.2.1. Différentes façons d'arrêter un programme

Les fonctions qui mènent par défaut à la terminaison du programme sont au nombre de trois. On trouve :

– `abort`. Cette fonction met fin au programme en déclarant celui-ci en situation d'erreur. Le message affiché est "*abnormal program termination*". Cette fonction n'est pas remplaçable par une de vos propres fonctions. Notez bien que la fonction `abort` peut être appelée explicitement dans votre code.

– `terminate`. Cette fonction met fin au programme en appelant, par défaut, la fonction `abort`. `terminate` est appelée lorsqu'une exception ne peut être traitée car aucun gestionnaire n'a été trouvé. De plus, il est possible de modifier son comportement en spécifiant le nom d'une de vos fonctions pour être appelée à la place de `terminate` (voir la section suivante). Cette fonction appelle, pour mettre fin au programme, la fonction `abort`.

– `unexpected`. Cette fonction met fin au programme en appelant la fonction `terminate`. Elle est appelée lorsqu'une exception ne peut être traitée car elle ne fait pas partie de la liste des exceptions valides définies pour la fonction qui

a levé l'exception (voir section 6.1.5.). Il est possible de modifier son comportement en spécifiant le nom d'une de vos fonctions pour être appelée à la place de `unexpected` (voir la section suivante). Cette fonction appelle, pour mettre fin au programme, la fonction `terminante`.

Notez par ailleurs qu'il est toujours possible d'arrêter un programme comme en langage C, c'est-à-dire en utilisant la fonction `exit` et en lui passant l'état de sortie de votre programme.

6.2.2. Remplacement de fonctions pour le traitement des erreurs

Comme nous l'avons précisé dans la section précédente, il est possible de changer certaines fonctions de terminaison d'un programme en cas de levée d'exceptions non gérées par le programme. À la lecture de la section précédente vous avez dû déduire que seule la fonction `abort` termine le programme : `terminate` s'exécute puis appelle `abort` (provoquant donc l'arrêt du programme), tandis que `unexpected` s'exécute puis appelle `terminate` (ce qui provoque donc la même chose). Cela implique donc que seule la fonction `abort` ne peut pas être redéfinie par l'utilisateur. Dans le cas des fonctions `terminate` et `unexpected` on utilise les fonctions `set_terminate` et `set_unexpected` en leur passant comme argument le nom de la fonction à appeler. Vous pouvez redéfinir l'une ou l'autre, voir les deux en même temps. Un exemple est indiqué ci-dessous.

```
1  #include <stdlib.h>
2  #include <stdio.h>
3  #include <eh.h>
4
5  void ma_fonction_terminate()
6  {
7   printf("Je suis dans ma fonction terminate!\n");
8  }
9
10 void ma_fonction_unexpected()
11 {
12  printf("Je suis dans ma fonction unexpected!\n");
13 }
14
15 void main()
16 {
17  set_terminate(ma_fonction_terminate);
18  set_unexpected(ma_fonction_unexpected);
19  unexpected();
20 }
```

Dans cet exemple, les nouvelles fonctions `terminate` et `unexpected` ne font qu'afficher un message à l'écran. L'appel de la fonction `unexepected` à la ligne 19 va donc provoquer l'affichage des messages des lignes 7 et 12 avant que le programme ne s'arrête dans l'état "*abnormal program termination*". Il est intéressant de noter que nous aurions pu également modifier la séquence `unexpected` → `terminate` → `abort` en ajoutant, par exemple, sur la ligne 13 la commande `exit(1)` ce qui aurait provoqué la sortie du programme dès l'appel à `unexpected` et sans que ni `terminate`, ni `abort` ne soient appelées.

6.3 UTILISATION DES EXCEPTIONS DANS LE DÉVELOPPEMENT DE PROGRAMME

Le mécanisme des exceptions a été très largement abordé dans ce chapitre ainsi que dans le chapitre 1. Il vous apparaît donc clairement comment cela fonctionne aussi bien sur un plan théorique que sur un plan pratique. De plus, il n'est plus besoin de vous expliquer en quoi il est très important d'intégrer les exceptions dans vos programmes dès leur conception. Dans l'annexe E je vous propose d'utiliser une classe, nommée `Cexception`, pour déclarer les objets levés lors d'une exception. Cette classe est vraiment basique, dans le sens où la seule information qu'elle permet de faire remonter lors d'une exception est une valeur numérique représentative de l'état de l'exception. Néanmoins, cette classe suffit à ce que vous puissiez déclarer des exceptions dans l'esprit "génie logiciel".

En réfléchissant bien à une utilisation plus poussée des exceptions, il est possible de construire une hiérarchie des exceptions. C'est-à-dire qu'en partant de la classe proposée dans l'annexe E vous pouvez construire plusieurs classes `Cexception` : une classe pour la levée d'exceptions suite à des erreurs mémoire (classe `Cexception_memoire`), une autre pour des erreurs de disque (classe `Cexception_disque`)... Idéalement ces classes seront définies comme classes héritées de la classe `Cexception` de l'annexe E (voir le chapitre 7 pour la définition de l'héritage). De même vous pouvez très bien regarder dans votre compilateur C++ car il peut très bien exister des classes `Cexception` déjà définies. Par exemple, dans la librairie *Microsoft Foundation Class* vous trouverez une classe `CException` de laquelle dérivent plusieurs autres classes comme `CMemoryException`, `CFileException`... Ces classes sont définies dans la librairie `afx.h`.

Chapitre 7

L'héritage

7.1 L'HÉRITAGE SIMPLE

L'héritage (également appelé dérivation) est un mécanisme de base de la programmation orientée objets qui permet de préciser des relations particulières entre classes. Ainsi, il est possible de spécifier qu'une classe `Cfille` hérite d'une classe `Cmere`, c'est-à-dire que la première constitue une *extension* de la seconde. Autrement dit, une classe hérite d'une autre quand elle en acquiert les propriétés (attributs et méthodes). On peut ainsi construire une classe sur la base d'une ou de plusieurs autres, en ajoutant ou en modifiant des propriétés des anciennes dans la nouvelle. Parfois, lorsqu'on est peu habitué à la programmation orientée objets, on a du mal à savoir quand utiliser l'héritage et quel est son apport. En réalité, la réponse à cette question se situe au niveau conceptuel : vous utilisez l'héritage quand le sens que vous donnez aux classes que vous créez vous l'impose. Autrement dit, sachant que le mécanisme d'héritage existe c'est à vous de décider d'un découpage en classes de votre programme qui l'utilise ! Prenons un exemple. Vous devez écrire un programme qui permet de gérer le personnel d'une entreprise. Lors de la phase de spécification et de conception, vous allez alors identifier une classe `Cpersonne` dont héritent les classes `Ccomptable`, `Csecretaire`... La classe possèdera des attributs représentant la date de naissance, la date d'embauche, etc. Ces attributs seront présents, par héritage dans chacune des classes filles. Vous voyez ? C'est le bon sens, au niveau spécification et conception, qui vous indique quand utiliser l'héritage ou pas.

Il existe deux formes d'héritage : l'héritage simple et l'héritage multiple. Dans l'héritage simple une classe dérivée n'a qu'une seule super-classe, ou classe mère. C'est le cas le plus simple que l'on puisse rencontrer. La classe dérivée apparaît comme une copie de sa super-classe avec les modifications apportées (ajout de propriétés). L'intérêt de l'héritage simple est de pouvoir concevoir tout ou partie d'un programme comme un ensemble de poupées russes : une classe en étendant une autre, elle prend plus de sens. L'héritage permet aussi de rendre plus simple l'utilisation d'une classe car celle-ci peut avoir peu d'attributs ou de méthodes spécifiques mais hériter de propriétés utilisées dans ces méthodes. Du coup, de l'extérieur de la classe "il semble" facile de l'utiliser sachant que tous les traitements complexes sont encapsulés dans les classes parentes.

L'héritage est également un outil qui permet de modéliser plus facilement la structure d'un programme puisqu'il donne du sens aux relations entre classes d'une même lignée.

7.1.1. Mise en œuvre

La déclaration d'héritage se fait lors de la déclaration de la classe dérivée (également appelée classe fille).

```
class nom_classe_fille : type_acces nom_classe_mere
```

Lors de la déclaration de la classe fille, il suffit donc de faire suivre son nom par celui de la classe mère dont on désire qu'elle dérive. Les noms des classes doivent être séparés par " :". On fait précéder le nom de classe mère par un modificateur d'accès qui influe sur les contrôles d'accès aux propriétés héritées dans la classe fille. La classe mère doit être connue à ce moment (généralement par inclusion de son fichier de déclaration). En voici un exemple.

```
1  class Cmere
2  {
3   private :
4           int iMERage;
5   ...
6  };
7
8  class Cfille : public Cmere
9  {
10  private :
11          float fFILargent;
12  ...
13 };
```

Dans cet exemple, l'héritage est `public` (nous verrons dans la section suivante ce que cela implique). Le tableau suivant récapitule les propriétés de chacune des classes de l'exemple.

classe `Cmere`	classe `Cfille`
`iMERage`	`iMERage`
...	`iFILargent`
	...

Ainsi, les objets de la classe `Cfille` possèdent les mêmes attributs et les mêmes méthodes que les objets de la classe `Cmere`, avec en plus les attributs et méthodes définies spécifiquement dans la classe `Cfille`.

7.1.2. Contrôles d'accès

La définition du type d'accès lors de l'héritage emploie les mêmes mots-clefs que pour les attributs et les méthodes d'une classe : `public`, `protected` et `private`. En fonction de la qualification, on a les situations suivantes :

(1) `public` : tous les contrôles d'accès de la classe mère sont conservés tels quels dans la classe fille.

(2) `protected` : tous les membres publics de la classe mère deviennent protégés dans la classe fille ; rien n'est changé pour les autres membres.

(3) `private` : tous les membres publics et protégés de la classe mère deviennent privés dans la classe fille. Par contre, les membres protégés de la classe mère restent accessibles aux membres et aux fonctions amies de la classe dérivée. Mais ils deviendront automatiquement privés lors d'un héritage future.

En l'absence d'un de ces mots-clefs lors de la déclaration d'héritage, le compilateur considère par défaut que vous faites un héritage privé !
Après héritage, les membres publics restent accessibles à l'utilisateur : que ces membres appartiennent à la classe fille ou à sa classe mère. Les membres protégés et les membres privés sont inaccessibles à l'utilisateur. Par contre les membres protégés de la classe mère sont accessibles à la classe fille. Elle les voit comme s'ils étaient définis par elle-même. Rappelons aussi que les membres protégés sont accessibles aux fonctions amies de la classe, puisque c'est la raison d'être de l'amitié. Les membres privés de la classe mère sont inacessibles à la classe fille.

Le tableau 7.1 récapitule les différentes situations d'héritage pouvant survenir. Pour chaque classe et chaque type d'héritage sont précisés les statuts des membres ainsi que leur accès interne et leur accès externe. On dit qu'on a accès en interne à un membre d'une classe si celui-ci est référençable à l'intérieur des méthodes de la classe (il n'y a pas d'erreur de compilation). Il y a un accès externe s'il est référençable par une fonction ou une méthode qui n'appartient pas à la classe.

Classe mère			Classe fille après héritage `public`		
Statut initial	Accès interne	Accès externe	Statut	Accès interne	Accès externe
`public`	Oui	Oui	`public`	Oui	Oui
`protected`	Oui	Non	`protected`	Oui	Non
`private`	Oui	Non	`private`	Non	Non

Classe fille après héritage `protected`			Classe fille après héritage `private`		
Statut	Accès interne	Accès externe	Statut	Accès interne	Accès externe
`protected`	Oui	Non	`private`	Oui	Non
`protected`	Oui	Non	`private`	Oui	Non
`private`	Non	Non	`private`	Non	Non

TAB. 7.1: Accessibilité des membres de la classe mère au niveau de la classe fille

Le tableau 7.1 récapitule donc l'accessibilité des membres d'une classe mère au niveau d'une classe fille. Vous pouvez facilement transposer ces résultats si plus d'un héritage est fait et obtenir ainsi, par exemple, l'accessibilité des composants de la classe mère et de la classe fille au niveau d'une classe petite-fille en fonction du type d'héritage réalisé. Néanmoins, plutôt que de connaître ce tableau par cœur, il est bien plus judicieux de retenir les règles suivantes qui vous permettent de le reconstruire.

Règle 1 : Le statut, dans la classe fille, d'un membre de la classe mère est donné par le qualificatif le plus restrictif (en terme de droits d'accès) entre le contrôle d'accès du membre et le type d'héritage. Ainsi, un membre déclaré en `private` dans la classe mère et qui subit un héritage `public` devient `private` dans la classe fille puisque `private` est plus restrictif que `public`.

Règle 2 : L'accès externe, à partir de la classe fille, d'un membre de la classe mère est donné par le statut du membre dans la classe fille. Ainsi, un membre de la classe mère qui reste `public` dans la classe fille est accessible, via la classe fille, à l'extérieur de la classe.

Règle 3 : L'accès interne, à partir de la classe fille, d'un membre de la classe mère est toujours possible sauf si ce membre avait le statut `private` dans la classe mère. C'est cette règle qui justifie toute la différence entre un membre `private` et un membre `protected`.

À toute règle il existe une exception ! Même aux règles qui précisent les conditions d'accès aux propriétés au sein d'une classe fille. **En effet, en langage C++ vous pouvez très bien faire en sorte qu'un membre `public` ou `protected` change de statut dans la classe fille, et cela quel que soit le type d'héritage que vous**

réalisez. Cela est possible en déclarant au sein de la classe fille ce membre avec le mot-clef `using`. Reprenons l'exemple de la section 7.1.1. et supposons que l'attribut `iMERage` soit déclaré en `protected` au sein de la classe `Cmere`. Son statut au sein de la classe `Cfille` est donc `protected`. Or, si vous déclarez, au sein de cette dernière, après le mot-clef `public` :

`using` Cmere : :iMERage ;

vous changerez le statut de cet attribut : il sera de type `public` au sein de la classe `Cfille`. Si vous aviez placé cette déclaration après un autre contrôle d'accès, le statut de l'attribut aurait été celui de ce contrôle d'accès. Ce mécanisme fonctionne également avec les méthodes, pour lesquelles il suffit d'utiliser la syntaxe suivante :

`using` nom_classe_mere : :nom_methode ;

Vous n'avez pas à faire apparaître la liste des paramètres de la méthode concernée.

7.1.3. Accès aux membres de la classe de base

Lors de l'héritage, la classe dérivée reçoit toutes les propriétés de la classe mère dont elle hérite. Ces propriétés sont nommables dans la classe fille comme si elles y avaient été définies (aux restrictions d'accès près). Reprenons l'exemple proposé précédemment que nous modifions légèrement.

```
1  class Cmere
2  {
3   protected :
4           int iMERage ;
5   ...
6  };
7
8  class Cfille : public Cmere
9  {
10  private :
11          float fFILargent ;
12  ...
13  public :
14           Cfille()
15           {
16                   iMERage=0 ;
17                   ...
18           }
19 };
20
21 void main()
22 {
23  Cfille FILV1 ;
```

```
24
25  FILV1.iMERage=10;
26  ...
27 }
```

Dans cet exemple, l'attribut `iMERage` est défini avec le contrôle d'accès `protected`. Il y a donc un accès interne possible au niveau de la classe fille, mais pas un accès externe. Cela implique donc que l'affectation réalisée dans le constructeur de la classe fille (ligne 16) est compilable. Ce n'est pas le cas de la ligne 25.

Il est également possible de définir une méthode ou un attribut dans la classe fille, avec le même nom que celui utilisé dans la classe mère. On crée ainsi une copie locale à la classe fille. On appelle cela de *l'occultation* et non pas de la surcharge (voir le chapitre 8 pour une présentation complète du mécanisme de surcharge). En effet, lorsque le compilateur rencontre un symbole (nom d'attribut ou nom de fonction) il cherche sa définition. Pour cela, il commence toujours par regarder dans l'objet s'il est défini avant de remontrer aux classes parentes. Ainsi, la copie locale est toujours vue avant le membre déclaré dans la classe parente. Si dans l'exemple précédent nous avions ajouté un attribut `float iMERage` avec le statut `public` dans la classe fille, la ligne 25 aurait compilé... puisque c'est la version `float` que le compilateur aurait modifié.

La question qui vient est alors : *Comment accéder à un membre occulté ?* La réponse est simple... il suffit de nommer complètement ce membre *en utilisant son nom long* ! Ainsi, pas d'ambiguïté possible pour le compilateur puisque vous lui avez indiqué clairement le membre à manipuler. Par exemple, si dans la fonction `main` précédente je veux référencer l'attribut `iMERage` de la classe mère je dois écrire :

```
FILV1.Cmere::iMERage=10;
```

Évidemment cela ne compilera pas puisque cet attribut est protégé. En tout cas vous avez utilisé le nom long pour le référencer.

7.1.4. Compatibilité avec la classe de base

Une classe fille est compatible avec chacun de ses ancêtres, puisqu'elle en est une extension. Cela se comprend d'autant mieux si l'on regarde la façon dont sont stockés en mémoire les objets d'une classe.

La figure 7.1 montre deux objets stockés en mémoire : un de la classe mère et un de la classe fille. Ce second possède donc les attributs[1] de la classe mère plus ceux qui lui sont spécifiques (attributs 4 et 5). **Pour l'objet de la classe fille, les**

1. En mémoire, seules les valeurs des attributs sont stockées au sein de l'objet. Il n'y a pas de copie du code des méthodes de la classe au sein des objets.

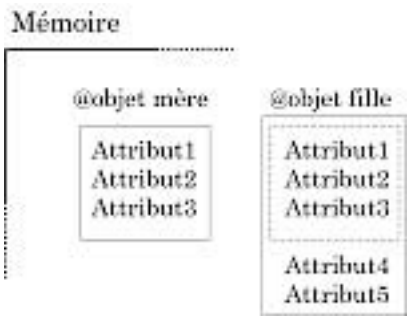


FIG. 7.1: Compatibilité d'une classe fille avec ses ancêtres

attributs venant de la classe mère sont stockés en premier. Ce point est très important car il permet de rendre compatible une classe fille avec sa mère. Ainsi une fonction qui attend en paramètre un objet de la classe mère peut très bien recevoir un objet de la classe fille puisque les attributs qu'elle est susceptible de manipuler se trouvent au début de l'objet comme dans la classe mère. Vous pouvez facilement imaginer ce que cela devient quand on a plusieurs héritages successifs : vous obtenez des poupées russes dont chacune est compatible avec tous ses ancêtres. Je vous suggère d'apprendre par cœur la règle suivante qui vous permettra de vous y retrouver.

> Là où la mère va la fille va, là où la fille passe la mère trépasse !

Voici un exemple concret dans lequel on suppose que la classe `Cfille` hérite de la classe `Cmere` (et peu importe le type d'héritage utilisé).

```
1  Cmere f(Cmere MERP1);
2  Cfille g(Cfille FILP1);
3
4  void main()
5  {
6   Cmere MERV1;
7   Cfille FILV2;
8
9   MERV1=FILV2; // Compile
10  FILV2=MERV1; // Ne compile pas
11
12  MERV1=f(FILV2); // Compile
13  FILV2=g(MERV1); // Ne compile pas
14
15  FILV2=f(FILV2); // Ne compile pas
```

```
16  MERV1=g(FILV2) ; // Compile
17  }
```

La ligne 9 compile puisqu'à droite du signe "=" est attendu un objet de type `Cmere`[2] et nous lui fournissons un objet de type `Cfille` : *là où la mère va, la fille va*. Donc cela compile. Par contre, cela n'est pas le cas de la ligne 10 puisqu'un objet de type `Cfille` est attendu à droite du "=" et nous lui fournissons un objet contenant moins d'attributs : il ne sait pas faire l'affectation complètement... *là où la fille passe, la mère trépasse*.

La ligne 12 compile puisque nous passons un objet de type `Cfille` en paramètre alors que c'est un objet de type `Cmere` qui est attendu. Par contre, la ligne 13 ne peut pas compiler pour la même raison que précédemment. Les objets retournés par les fonctions `f` et `g` sont du même type que ceux attendus à gauche du signe "=".

Les lignes 15 et 16 sont intéressantes puisqu'en fait dans les deux cas les appels aux fonctions compilent. Ce qui ne marche pas sur la ligne 15 c'est de retourner un objet de type `Cmere` en sortie de fonction `f` alors qu'à gauche du signe "=" est attendu un objet de type `Cfille`. Dans ce cas, c'est donc l'affectation qui ne sera pas compilable.

7.1.5. Appels des constructeurs et des destructeurs

Lors de la création ou de la destruction d'un objet d'une classe dérivée, l'appel des constructeurs et des destructeurs de la classe fille et de la classe mère est réalisé automatiquement :

– pour les constructeurs : l'appel d'un constructeur d'une classe fille n'est effectivement réalisé qu'après l'appel du constructeur de sa classe mère. On applique la même règle sur la classe mère si celle-ci est également héritière d'une autre classe. Cela revient à dire que le compilateur programme l'appel de tous les constructeurs des classes ancêtres (*phase d'identification* des constructeurs à appeler) d'une classe en commençant par le plus ancien avant de réaliser les appels effectifs (*phase d'appel*). La figure 7.2 schématise ce mécanisme.

– pour les destructeurs : les appels aux destructeurs sont réalisés dans l'ordre inverse des constructeurs, c'est-à-dire qu'après l'appel du destructeur d'une classe, on appelle celui de sa classe mère. La même règle est appliquée sur la classe mère si elle est héritière.

Par défaut, lors de l'appel des constructeurs, c'est le constructeur par défaut des classes parentes qui est appelé car bien souvent aucune spécification contraire n'est donnée lors de la création des classes. Néanmoins, cela peut être changé puisque vous avez la possibilité de transmettre des arguments au constructeur de la classe mère d'une classe. Lors de la définition du constructeur de la classe fille, on appelle

2. Le compilateur cherche toujours à mettre l'objet de droite dans le type de l'objet de gauche.

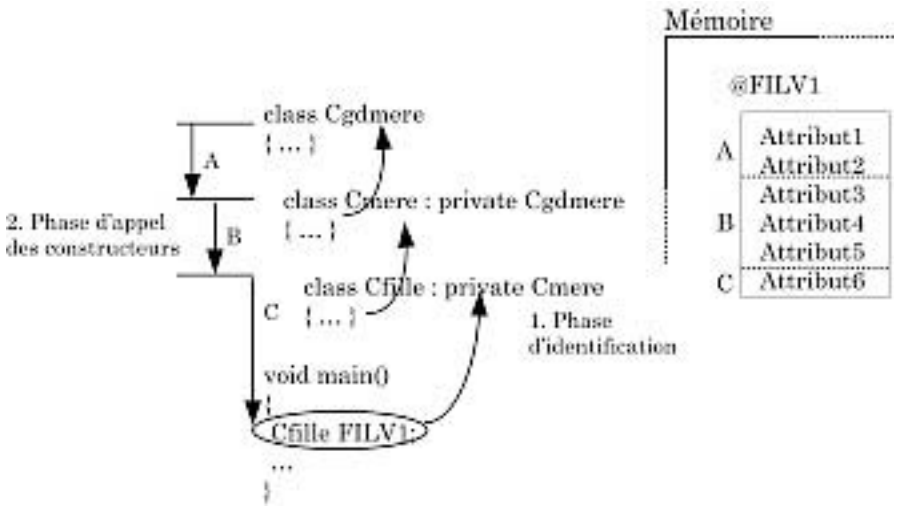


FIG. 7.2: Mécanisme d'appel des constructeurs dans le cadre de l'héritage

explicitement le constructeur de la classe mère avec les paramètres nécessaires. L'exemple suivant montre le passage d'un paramètre au constructeur de la classe mère lors de la définition d'un constructeur de la classe fille.

```
1  class Cmere
2  {
3   ...
4   Cmere(int iP1);
5  };
6
7  class Cfille : private Cmere
8  {
9   ...
10  Cfille(int iP1,float fP2) : Cmere(iP1);
11 };
12
13 void main()
14 {
15  Cfille FILV1(5,6.7);
16  ...
17 }
```

Cet exemple montre bien comment préciser le constructeur de la classe mère à appeler (ligne 10) lors de l'écriture du constructeur de la classe fille. Ainsi, la création de l'objet `FILV1`, sur la ligne 15, va provoquer l'identification dans un premier

temps des constructeurs à appeler. La ligne 15 stipule que pour la classe `Cfille` c'est le constructeur à deux arguments qui doit être utilisé. La lecture de la ligne 10 indique au compilateur qu'il doit appeler, pour la classe `Cmere`, le constructeur à un argument de type `int`. Comme il n'y a plus de classe parente, le compilateur réalise l'appel effectif du constructeur de la classe `Cmere` avant d'appeler celui de la classe `Cfille`.

7.1.6. Cas du constructeur de recopie

Dans la section 7.1.5. nous avons vu le mécanisme d'appel des constructeurs/destructeurs lorsqu'un objet d'une classe fille était créé/détruit. Il y a cependant une exception à la règle d'appel des constructeurs : le cas de l'appel au constructeur de recopie.

Rappelez-vous que l'appel d'un constructeur de recopie se fait principalement dans les deux cas suivants :

(1) On initialise un objet avec un objet de la même classe (ou compatible par héritage).

(2) On transmet la valeur d'un objet en argument ou en retour d'une fonction.

Si on se trouve dans une de ces situations pour une classe fille, on va chercher à appeler son constructeur de recopie. Deux cas peuvent se présenter :

(1) Ce contructeur n'est pas défini explicitement par l'utilisateur : on appelle celui définit par défaut au sein du compilateur (une recopie membre à membre des attributs de la classe fille uniquement sera réalisée). On remonte au niveau de la classe mère et on se pose la question de savoir s'il existe un constructeur de recopie défini.

(2) Ce constructeur est défini explicitement par l'utilisateur : il y a appel automatique du constructeur de recopie de la classe fille. Celui de la classe mère ne sera pas automatiquement appelé, contrairement à ce qui est fait dans le cas d'un constructeur autre. Cela veut dire que le constructeur de recopie de la classe fille doit prendre en charge la recopie de tous ses attributs, y compris ceux émanant de ses ancêtres. Il reste néanmoins possible d'appeler un constructeur de la classe mère en utilisant le mécanisme de la remontée d'information entre constructeurs (cf. l'exemple de la section 7.1.5.).

Ainsi, si vous définissez un constructeur de recopie dans une classe vous savez que la phase d'identification sera interrompue par cette classe et que l'initialisation des attributs des classes parentes doit être explicitement considérée.

7.2 L'HÉRITAGE MULTIPLE

Dans l'héritage multiple une classe fille possède plusieurs classes mères. Elle hérite de toutes leurs propriétés. Cela veut donc dire qu'avant d'aller plus loin dans la lecture de cette partie, vous devez avoir bien maîtrisé les concepts liés à l'héritage simple. Typiquement, l'héritage multiple est beaucoup moins utilisé que l'héritage simple car il implique que votre logiciel possède une architecture complexe. Néanmoins, il peut avoir son utilité.

Reprenons l'exemple cité en introduction de ce chapitre, celui du logiciel de gestion du personnel d'une entreprise. Supposez que vous ayez à créer une classe permettant de gérer des technico-commerciaux à partir d'une classe `Ccommercial` et `Ctechnicien`. Vous allez donc créer une classe `Ctechcom` qui hérite des classes `Ccommercial` et `Ctechnicien`. Là encore, c'est le bon sens qui vous a indiqué l'utilisation de l'héritage multiple. Néanmoins, d'un point de vue purement génie logiciel on préfèrera éviter au maximum l'utilisation de l'héritage multiple qui tend à complexifier le codage du logiciel.
Nous allons maintenant voir comment mettre en œuvre l'héritage multiple.

7.2.1. Mise en œuvre

La déclaration d'héritage multiple est similaire à la déclaration d'héritage simple.

`class` nom_classe_fille : type_acces nom_classe_mere1, type_acces nom_classe_mere2, ...

Lors de la déclaration de la classe fille, il suffit donc de faire suivre son nom par celui des classes mères dont on désire qu'elle dérive. On fait précéder les noms des classes mères par des modificateurs d'accès qui influent sur les contrôles d'accès aux propriétés héritées dans la classe fille. Les classes mères doivent être connues à ce moment (généralement par inclusion des fichiers de déclaration).

Notez bien que les règles énoncées dans la section 7.1.2. sur le statut et l'accessibilité des propriétés dans la classe fille sont toujours valables.

7.2.2. Membres homonymes dans les classes mères

L'utilisation de l'héritage multiple peut faire apparaître des propriétés homonymes dans la classe fille. Supposez que dans les classes `Cmere1` et `Cmere2` vous ayez deux attributs qui portent le même nom : que se passe-t-il au niveau de la classe fille ? Y a-t-il une erreur de compilation du programme ? Comme vous pouvez le supposer vous aurez systématiquement une erreur de compilation si vous n'utilisez pas le nom long des méthodes ou attributs en double. En effet, le compilateur ne saura pas quelle méthode ou quel attribut vous référencez et ne sera donc pas capable de savoir si vous y avez accès ou pas. L'exemple suivant illustre des cas de figure qui marchent.

```
1  #include <stdio.h>
2
3  class Cmere1
4  {
5   protected :
6           int iA1 ;
7   public :
8           int Lire_A1() {return iA1 ;}
9           void Ajouter_A1(int iP1) {iA1+=iP1 ;}
10 };
11
12 class Cmere2
13 {
14  public :
15          int iA1 ;
16  public :
17          int Lire_A1() {return iA1 ;}
18          void Ajouter_A1() {iA1+=1 ;}
19 };
20
21 class Cfille : private Cmere1, public Cmere2
22 {
23  public :
24          Cfille()
25          {
26           Cmere1 : :iA1=0 ;
27           Cmere2 : :iA1=1 ;
28          }
29 };
30
31 void main()
32 {
33  Cfille FILV1 ;
34
35  printf("Valeur de l'attribut Cmere2 : :iA1 : %d\n",FILV1.Cmere2 : :Lire_A1()) ;
36  FILV1.Cmere2 : :Ajouter_A1() ;
37 }
```

En règle générale on évitera que ce cas de figure se produise, c'est-à-dire qu'on évitera d'avoir des attributs ou des méthodes héritées en exemplaires multiples au sein des classes filles. Notez bien que si vous utilisez les conventions de nommage présentées dans la section 1.4.2., il y très peu de chance que cela se produise.

Un autre cas de figure plus gênant peut survenir lorsque vous utilisez l'héritage multiple. *Supposez dans l'exemple précédent que les classes* `Cmere1` *et* `Cmere2`

héritent toutes les deux d'une même classe `Cmere0`. Que se passe-t-il au niveau de la classe `Cfille` ? Et bien la même chose que précédemment, à savoir que vos objets de la classe fille possèderont en double les attributs de la classe `Cmere0`. Donc, pour référencer ces attributs vous devrez utiliser leur nom long... mais en mettant comme nom de classe de rattachement le nom de la classe `Cmere1` ou `Cmere2` et non pas `Cmere0` ! Regardez comment est écrit le constructeur de la classe `Cfille` dans l'exemple suivant.

```
1  #include <stdio.h>
2
3  class Cmere0
4  {
5   public :
6            float fM0A1 ;
7  };
8
9  class Cmere1 : protected Cmere0
10 {
11  protected :
12           int iM1A1 ;
13  public :
14           int M1Lire_A1() {return iM1A1 ;}
15           void M1Ajouter_A1(int iP1) {iM1A1+=iP1 ;}
16 };
17
18 class Cmere2    : protected Cmere0
19 {
20  public :
21           int iM2A1 ;
22  public :
23           int M2Lire_A1() {return iM2A1 ;}
24           void M2Ajouter_A1() {iM2A1+=1 ;}
25 };
26
27 class Cfille : private Cmere1, public Cmere2
28 {
29  public :
30           Cfille()
31           {
32            iM1A1=0 ;
33            iM2A1=1 ;
34            Cmere1 : :fM0A1=2 ; // On référence l'attribut venant de Cmere0 via Cmere1
35            Cmere2 : :fM0A1=3 ; // On référence l'attribut venant de Cmere0 via Cmere2
36           }
37 };
```

```
38
39 void main()
40 {
41  Cfille FILV1;
42
43  printf("Valeur de l'attribut Cmere2 : :iA1 : %d\n",FILV1.M2Lire_A1());
44  FILV1.M2Ajouter_A1();
45 }
```

Dans le cas précédent de duplication par héritage, il existe une possibilité offerte par le langage C++ et qui permette d'éviter cette duplication. Il suffit de déclarer que l'héritage entre la classe mère et la classe fille est qualifié de virtuel. La syntaxe est la suivante :

`class` nom_classe_fille : type_acces `virtual` nom_classe_mere

Ainsi, tout objet d'une classe héritant de la classe fille ne possèdera qu'une seule copie de la classe mère. En réalité, **il est nécessaire que toutes les classes filles héritant de la classe mère, et dont hérite la classe petite-fille, doivent déclarer leur héritage avec la classe mère en virtuel**. Voici l'exemple précédent modifié pour éviter d'avoir une duplication des attributs de la classe `Cmere0` au sein de la classe `Cfille`.

```
1  #include <stdio.h>
2
3  class Cmere0
4  {
5   public :
6            float fM0A1;
7  };
8
9  class Cmere1 : protected virtual Cmere0
10 {
11  protected :
12           int iM1A1;
13  public :
14           int M1Lire_A1() {return iM1A1;}
15           void M1Ajouter_A1(int iP1) {iM1A1+=iP1;}
16 };
17
18 class Cmere2   : protected virtual Cmere0
19 {
20  public :
21           int iM2A1;
22  public :
23           int M2Lire_A1() {return iM2A1;}
```

```
24          void M2Ajouter_A1() {iM2A1+=1;}
25 };
26
27 class Cfille : private Cmere1, public Cmere2
28 {
29  public :
30          Cfille()
31          {
32           iM1A1=0;
33           iM2A1=1;
34           Cmere1::fM0A1=2; // On référence l'unique attribut venant de Cmere0
35           Cmere2::fM0A1=3; // On référence l'unique attribut venant de Cmere0
36           printf("Valeur de l'attribut fM0A1 : %f\n",Cmere1::fM0A1);
37          }
38 };
39
40 void main()
41 {
42  Cfille FILV1;
43
44  printf("Valeur de l'attribut Cmere2 : :iA1 : %d\n",FILV1.M2Lire_A1());
45  FILV1.M2Ajouter_A1();
46 }
```

L'affichage suite à l'exécution de cet exemple est le suivant :

```
Valeur de l'attribut fM0A1 : 3.0000000
Valeur de l'attribut Cmere2 : :iA1 : 1
```

La première ligne affichée (elle correspond à la ligne 36 de l'exemple) montre que la ligne 35 du code a modifié le même attribut que celui référencé dans la ligne 34. Donc, il n'y a qu'une seule copie des attributs de la classe `Cmere0` au sein de l'objet `FILV1` de la classe `Cfille`.
Par contre, il suffit de retirer sur la ligne 9 **ou** sur la ligne 18 le mot-clef `virtual` pour que la duplication des attributs de la classe `Cmere0` ait lieu.

7.2.3. Constructeurs et destructeur

L'appel des *constructeurs* se fait toujours selon le même principe que dans le cas de l'héritage simple : il y a tout d'abord une phase d'identification des constructeurs à appeler, en partant de la classe fille, puis une phase d'appel effectif des constructeurs, en partant des classes les plus anciennes. Lorsqu'une classe a plusieurs classes mères, l'appel des constructeurs des classes mères se fait dans l'ordre défini lors de la déclaration d'héritage dans l'interface de la classe fille. L'appel des *destructeurs* se fait selon le même principe mais en commençant par la classe fille et en remontant dans ses ancêtres.

Et maintenant que se passe-t-il si vous avez, par héritage, la présence d'un objet en double ? Reprenons l'exemple de la section précédente dans lequel : `Cmere1` et `Cmere2` héritent de `Cmere0` et la classe `Cfille` hérite de `Cmere1` et `Cmere2`. Donc, `Cfille` possède par défaut les attributs de la classe `Cmere0` en double... sauf si vous avez fait un héritage virtuel pour les classes `Cmere1` **et** `Cmere2`. Supposons que ce soit le cas. Comment se passe alors l'appel des constructeurs, notamment ceux de la classe `Cmere0` ? Et bien tout simplement, seul le constructeur de recopie ou le constructeur par défaut peuvent être appelés pour l'initialisation des objets de la classe `Cfille`. Ainsi, une syntaxe comme :

```
Cmere1(liste_parametres) : Cmere0(liste_parametres)
```

n'aura aucun effet et provoquera l'appel au constructeur par défaut pour la classe `Cmere0`. Pour réaliser l'appel à un constructeur spécifique il faut écrire cette ligne non pas au niveau des classes `Cmere1` et `Cmere2` mais au niveau de la classe `Cfille` directement.

```
Cfille(liste_parametres) : Cmere0(liste_parametres)
```

Chapitre **8**

La surcharge

8.1 LA SURCHARGE DE FONCTIONS ET DE MÉTHODES

La surcharge de fonctions/méthodes est un mécanisme du langage C++ permettant pour une fonction/méthode d'avoir plusieurs exécutions possibles en fonction du contexte de son appel. On définit ici le contexte de l'appel par le type des arguments passés en paramètre de la fonction/méthode. En fait, on est autorisé à utiliser plusieurs fois le même nom pour des fonctions/méthodes différentes, à condition que leur prototype offre un moyen de les différencier. Notez bien que seule la liste des paramètres passés à la fonction/méthode permet de la différencier d'une autre fonction/méthode ayant le même nom : le type de l'objet de retour, s'il y en a un, ne permet pas de les différencier.

Ce mécanisme vous évoque-t-il quelque chose ? Et bien en fait, vous avez déjà rencontré la surcharge de fonctions. Ou plutôt la surcharge de méthodes... lorsque vous avez vu les constructeurs ! En effet, deux constructeurs d'une même classe possèdent le même nom et constituent bien une surcharge de la même méthode.

Voici un exemple qui vous permettra d'intuiter comment se passe le choix d'une version surchargée par le compilateur en fonction du contexte d'appel.

```
1  #include<stdio.h>
2
3  int NombreAuCarre(int iP1)
4  {
5   printf("Exécution de la version 1\n");
6   return(iP1*iP1);
7  }
8
9  double NombreAuCarre(double fP1)
10 {
11  printf("Exécution de la version 2\n");
12  return(fP1*fP1);
13 }
14
15 void main()
16 {
17  printf("Valeur du carré : %d\n",NombreAuCarre(5));
18  printf("Valeur du carré : %lf\n",NombreAuCarre(5.4));
19 }
```

Comme vous pouvez le deviner l'affichage à l'écran va être le suivant :

```
Exécution de la version 1
Valeur du carré : 25
Exécution de la version 2
Valeur du carré : 29.160000
```

La détermination de ces versions a pu être réalisée par le compilateur grâce aux valeurs numériques passées en paramètre sur les lignes 17 et 18.

Nous allons voir maintenant plus en détail quelles sont les règles qui permettent au compilateur de décider de la version surchargée à appeler. Nous allons notamment distinguer le cas des fonctions/méthodes à un argument et le cas des fonctions/méthodes admettant plusieurs arguments.

8.1.1. Cas des fonctions/méthodes à un argument

Dans le cas de fonctions à un argument le choix de la bonne version de la fonction surchargée par le compilateur se fait en trois niveaux. Partant de l'appel de la fonction réalisé par l'utilisateur, le compilateur regarde les versions dont il dispose.

– Niveau 1 : *Correspondance exacte*. L'appel est effectué avec une valeur dont le type a été prévu par l'utilisateur dans une des versions surchargées. On tient compte des attributs de signe[1].

1. `char` est différent de `signed char` par exemple.

– Niveau 2 : *Promotion numérique*. Il n'existe pas de version dont le type de l'argument est celui du paramètre passé lors de l'appel. Le compilateur cherche alors une version dont l'argument est d'un type compatible "plus large" que celui passé en paramètre : (`short`→`int`), (`char`→`int`), (`float`→`double`)... Le terme "*type plus large*" fait référence ici à l'ensemble des valeurs qu'il peut prendre : ainsi un type A est plus large qu'un type B si l'ensemble des valeurs du type B est inclus dans celui du type A.
– Niveau 3 : *Conversions standard*. Si aucune version surchargée n'a pu être choisie aux niveaux précédents, le langage C++ va réaliser automatiquement des conversions entre types de base, pour lesquelles la conversion n'a pas besoin d'être précisée (*conversions implicites*). Ce cas permet de convertir vers un type "moins large", avec perte éventuelle d'information, mais ne concerne que des types de base. Si aucune version n'est candidate le compilateur va chercher à réaliser des *conversions définies par l'utilisateur*, c'est-à-dire des conversions entre/vers des objets de classes. Par ailleurs, passer un objet d'une classe fille alors qu'un objet d'une classe mère est attendu est considéré comme une conversion standard (plus précisément une *conversion définie par l'utilisateur*).

Notez bien qu'ici vous pouvez avoir deux situations d'erreur. Tout d'abord, si lors du niveau 2 ou du niveau 3 le compilateur identifie plusieurs versions surchargées qui peuvent être appelées alors à la compilation vous aurez le message d'erreur "*Appel ambigu de fonction surchargée*" sur la ligne où vous réalisez l'appel (ligne 17 dans l'exemple ci-dessous).

```
1  #include<stdio.h>
2
3  float NombreAuCarre(float fP1)
4  {
5   printf("Exécution de la version 1\n");
6   return(fP1*fP1);
7  }
8
9  double NombreAuCarre(double dP1)
10 {
11  printf("Exécution de la version 2\n");
12  return(dP1*dP1);
13 }
14
15 void main()
16 {
17  printf("Valeur du carré : %lf\n",NombreAuCarre(5));
18 }
```

L'exemple précédent ne compile pas car l'argument passé à la fonction `NombreAuCarre` sur la ligne 17 est assimilé à un `int`. Puisqu'il n'existe pas de version surchargée acceptable au titre de la correspondance exacte (niveau 1), le compilateur va chercher une version ayant un paramètre d'un type plus large (niveau 2). Or, il en existe deux : une prenant un paramètre de type `float` et une autre prenant un paramètre de type `double`. Cela conduit donc à une ambiguïté.
La seconde situation d'erreur possible est si, à l'issue du niveau 3, aucune version surchargée ne peut être choisie pour l'appel que vous avez fait. Dans ce cas vous aurez, à la compilation, le message d'erreur "*Aucune des fonctions surchargées ne peut convertir votre paramètre dans un de ceux possibles*".

8.1.2. Différenciation des prototypes

Avant d'aller plus loin et d'aborder le cas des fonctions à plusieurs arguments, il est nécessaire de présenter quelques cas bien particuliers qui font que le compilateur considère deux versions surchargées différentes ou pas.

(1) On ne peut différencier deux arguments selon une référence : un type et une référence à un type sont considérés comme identiques.

(2) Un pointeur sur un type A et un autre sur un type B sont considérés comme différents, bien que cela soit dans les deux cas des adresses mémoires.

(3) Un objet d'un type A et un autre d'un type défini par un `typedef` du type A sont identiques.

(4) Un objet constant (`const`) d'un type A est considéré comme un objet d'un type A : le mot-clef `const` n'est pas pris en compte lors du choix d'une version surchargée. **Ce n'est pas vrai lorsque l'argument n'est plus un objet mais une référence ou un pointeur !**

(5) Un objet d'un type énumeratif (`enum`) et un objet de type `int` sont considérés comme différents (bien qu'ils aient la même nature).

(6) Un tableau d'éléments d'un type A et un pointeur vers un type A sont considérés comme identiques (le tableau est transmis par l'adresse de son premier élément).

(7) Si vous passez une valeur numérique de type réel (par exemple, la valeur 5.4) elle est considérée comme un `double` et non comme un `float`.

Revenons à présent sur le cas des arguments passés par référence et considérons une version modifiée du cas de la fonction `NombreAuCarre`.

```
1  #include<stdio.h>
2
3  long int NombreAuCarre(long int &liP1)
4  {
5   printf("Exécution de la version 1\n");
6   return(liP1*liP1);
```

```
7  }
8
9  double NombreAuCarre(double dP1)
10 {
11  printf("Exécution de la version 2\n");
12  return(dP1*dP1);
13 }
14
15 void main()
16 {
17  printf("Valeur du carré : %lf\n",NombreAuCarre((long int)5));
18 }
```

Vous vous attendez à ce que ce soit la première version qui soit appelée ? Et bien non, en réalité c'est la seconde qui le sera bien que nous ayons précisé sur la ligne 17 une conversion explicite vers un objet de type `long int`. Surpris ? En fait cela est parfaitement compréhensible (mais il fallait y penser !) si l'on a assimilé les principes du passage par référence. En effet, un de ces principes stipule que c'est l'adresse de l'objet passé en paramètre qui est envoyé à la fonction ce qui implique que les modifications faites dans la fonction sont réalisées sur cet objet et non sur une copie. Or, ici, vous passez une valeur numérique. Donc, à la compilation, un emplacement mémoire temporaire va être alloué et la valeur 5 va lui être affectée. C'est l'adresse de cet emplacement temporaire qui va être passé à la fonction `NombreAuCarre`. Nous arrivons donc à une contradiction : les modifications faites sur l'argument ne seront pas accessibles à la sortie de la fonction puisque cet argument est stocké dans un emplacement temporaire. Vous n'auriez pas passé une valeur numérique mais une variable la situation aurait été différente. Ainsi, le compilateur va ignorer, dans ce cas précis une des deux versions. Il n'en restera plus qu'une : celle prenant un argument de type `double`, qui sera choisi au titre du niveau 2.

8.1.3. Cas des fonctions/méthodes à plusieurs arguments

Le cas de fonctions ou méthodes à plusieurs arguments est un peu plus complexe. Il suit les remarques énoncées dans la section 8.1.2. et reprend les niveaux de recherche indiqués dans la section 8.1.1. Le tout dans un processus un peu plus compliqué, présenté sous forme du pseudo-algorithme suivant.

(1) Existe-t-il une version surchargée dont le type de chaque argument correspond exactement à ceux passés lors de l'appel ? Si la réponse est positive alors le compilateur choisit cette version, sinon il continue son processus de choix.

(2) Pour chaque argument en position i, on définit l'ensemble E_i des versions surchargées pouvant être appelées en appliquant les niveaux 1, 2 ou 3.

(3) L'ensemble E est l'ensemble intersection des ensembles E_i. Éliminer de E les versions dominées à cause de *conversions définies* (niveau 3).

(4) Détermination de la fonction selon la procédure suivante :

Si (il existe une version `V` dans E dont tous les arguments sont choisis soit par le niveau 1, soit par le niveau 2 et `V` est unique) Alors
  Choisir `V`
Sinon
  Si (E ne contient qu'une seule version candidate `V`) Alors
    Choisir `V`
  Sinon
    Si (E ne contient pas de version) Alors
      Erreur : *Appel ambigu de fonction surchargée*
    Sinon
      Erreur : *Aucune des fonctions surchargées ne peut être appelée*
    Fin Si
  Fin Si
Fin Si

La notion de dominance utilisée dans le pseudo-algorithme précédent est liée à la priorité qu'accorde le compilateur, au sein du niveau 3, aux conversions standards par rapport aux conversions définies. Ainsi, le compilateur préfère accorder la priorité à une dégration numérique (donc avec perte d'information) plutôt que de réaliser une conversion vers une classe utilisateur (conversion définie).
Voici, pour illustrer, l'exemple de la fonction `Multiplier` qui permet de multiplier entre eux deux nombres.

```
1  #include<stdio.h>
2
3  float Multiplier(int iP1, float fP2)
4  { // Version V1 de la fonction
5   printf("Exécution de la version 1\n");
6   return((float)iP1*fP2);
7  }
8
9  double Multiplier(double dP1, int iP2)
10 { // Version V2 de la fonction
11  printf("Exécution de la version 2\n");
12  return(dP1*(double)iP2);
13 }
14
15 void main()
16 {
17  float fV1=5;
18  int iV2=6.4;
```

```
19  printf("Valeur de la multiplication : %lf\n",Multiplier(fV1,iV2));
20 }
```

L'appel réalisé sur la ligne 19 conduit à l'application suivante du pseudo-algorithme précédent :

(1) Il n'existe pas de version de `Multiplier` prenant un `int` en premier argument et un `float` en second argument.

(2) $E_1 = \{\mathtt{V}_1, \mathtt{V}_2\}$ et $E_2 = \{\mathtt{V}_1, \mathtt{V}_2\}$.

(3) $E = \{\mathtt{V}_1, \mathtt{V}_2\}$.

(4) Seule la version 2 a des arguments qui correspondent selon les niveaux 1 et 2 à l'appel réalisé. Le premier argument de la version 1 nécessite une dégradation numérique de `float` vers `int`. C'est donc la seconde version qui va être appelée.

Voici un second exemple qui illustre la notion de dominance. On suppose ici l'existence d'une classe `Cfraction` qui possède un constructeur à un argument de type `float`.

Comme nous le verrons dans la section 8.3.3., cela permet au compilateur de créer des objets temporaires initialisés à partir d'une valeur réelle, donc de réaliser des conversions définies par l'utilisateur de `float` vers `Cfraction`.

```
1  class Cfraction
2  {
3   public :
4            Cfraction(float fP1) {...}
5   ...
6  };
7
8  float Multiplier(Cfraction FRCP1, float fP2)
9  { // Version V1 de la fonction
10  printf("Exécution de la version 1\n");
11  return(...);
12 }
13
14 int Multiplier(int iP1, int iP2)
15 { // Version V2 de la fonction
16  printf("Exécution de la version 2\n");
17  return(iP1*iP2);
18 }
19
20 void main()
21 {
22  float fV1=5.7;
```

```
23  int iV2=6;
24  printf("Valeur de la multiplication : %d\n",Multiplier(fV1,iV2));
25 }
```

L'application du pseudo-algorithme de choix va mettre en évidence la notion de *dominance*.

(1) Il n'existe pas de version de `Multiplier` prenant un `float` en premier argument et un `int` en second argument.

(2) $E_1 = \{\mathtt{V}_1, \mathtt{V}_2\}$ et $E_2 = \{\mathtt{V}_1, \mathtt{V}_2\}$.

(3) $E = \{\mathtt{V}_1, \mathtt{V}_2\}$.

(4) Dominance : aucune fonction ne peut être choisie uniquement par les niveaux 1 ou 2. La première version nécessite une conversion standard (plus précisemment une conversion définie) pour le premier argument et une promotion numérique pour le second. La seconde version implique une dégradation (conversion standard) sur le premier argument et une correspondance exacte sur le second. Pourtant, la seconde version domine la première car le compilateur préfère faire une dégradation numérique qu'une conversion définie (conversion du `float` en `Cfraction`). Ainsi, on obtient $E = \{\mathtt{V}_2\}$ et la seconde version est choisie par le compilateur car c'est la seule dans E.

8.1.4. Surcharge de fonctions membres

La surcharge s'applique également aux méthodes d'une classe. Il faut cependant remarquer que la définition d'une méthode ayant la même interface dans deux classes parentes n'est pas une surcharge mais une occultation. En effet, une classe définit la portée de ses propres méthodes. Ainsi, si une méthode `f` est définie dans la classe mère et dans une classe fille, il n'y a pas de choix possible : selon l'objet à partir duquel vous faite l'appel de `f` c'est l'une ou l'autre qui sera appelée.
Pour appeler la version `f` de la classe mère à partir d'un objet de la classe fille, il vous faut donc utiliser son nom long, *i.e.* utiliser l'opérateur de résolution de portée. Par exemple, `fille.Cmere::f()`.

8.2 LA SURCHARGE D'OPÉRATEURS

En langage C++ il est possible de redéfinir le comportement des opérateurs existants, par exemple : +, -, *... Cela se fait en surchargeant ces opérateurs, c'est-à-dire en définissant leur comportement quand les opérandes ont des types particuliers. Évidemment, il ne vous est pas possible de redéfinir n'importe quoi, par exemple il n'est pas possible de redéfinir l'addition de deux entiers. La surcharge d'opérateurs à un sens lorsqu'au moins une des opérandes de l'opérateur concerné est un objet d'une classe.

8.2.1. Mécanisme

Nous allons maintenant voir les bases de la surcharge d'opérateurs.
Par convention, un opérateur `op` reliant deux opérandes `a` et `b` est vu comme une fonction prenant en argument un objet de type `a` et un objet de type `b`. Ainsi, l'écriture `a op b` est traduite en langage C++ par un appel de fonction `op(a,b)`. Évidemment ce travail de "réécriture" est transparent pour le programmeur et est réalisé à la compilation. Enfin un opérateur est réprésenté par un nom composé du mot-clef `operator` suivi de l'opérateur (par exemple : "+" a pour nom `operator+()`). Ainsi, l'écriture `5+4` est équivalente à l'appel de fonction `operator+(5,4)`. Ce mécanisme a un rôle primordial dans la compréhension de la surcharge d'opérateurs. Notamment, regardez bien la façon dont l'exemple précédent a été ramené à un appel de fonction : l'ordre des paramètres de la fonction `operator+` est le même que l'ordre des opérandes de l'opérateur +. En effet, *l'ordre des opérandes définit l'ordre des paramètres de la fonction.*

La surcharge d'un opérateur peut être définit en C++ comme une fonction membre d'une classe. En procédant ainsi, la méthode surchargeant l'opérateur possède un argument de moins que le nombre d'opérandes de celui-ci. Supposons que la variable `FRCfraction1` soit un objet de la classe `Cfraction`, alors l'opération

```
FRCfraction1 + 5
```

se traduit par l'appel de fonction

```
FRCfraction1.operator+(5)
```

Il faut donc que dans la classe `Cfraction` se trouve définie une méthode `operator+(int)` ... avec un seul argument alors que l'opérateur + est un opérateur binaire.

Prenons l'exemple ci-dessous pour préciser encore un peu plus le mécanisme de la surcharge d'opérateurs. On suppose évidemment que la définition de la surcharge de la ligne 4 compile correctement bien qu'elle ne soit pas donnée ici.

```
1  class Cfraction
2  {
3   public :
4            Cfraction operator+(int iP1);
5  };
6
7  void main()
8  {
9   Cfraction FRCfraction1;
10  FRCfraction1=FRCfraction1 + 5; // Cela compile
11  FRCfraction1=5 + FRCfraction1; // Cela ne compile pas
12 }
```

La ligne 11 compile et correspond au cas de figure que nous avons déjà évoqué un peu plus haut dans cette section. La ligne 12 ne compile pas quant à elle. Pourquoi ? Tout simplement à cause de l'ordre des opérandes : `5 + FRCfraction1` est en quelque sorte équivalent à "`int.operator+(Cfraction)`" or il n'y a pas ici de surcharge de l'opérateur + défini comme cela. Vous constatez donc que l'ordre des opérandes est loin d'être anodin.

Cet exemple soulève une dernière question : comment faire en sorte que la ligne 12 compile ? Et bien il est nécessaire de définir une fonction (donc qui n'appartient pas à la classe `Cfraction`) prenant deux arguments : le premier étant de type `int` et le second de type `Cfraction`. Si cette surcharge doit avoir accès à des éléments privés de la classe `Cfraction` alors vous devez définir cette surcharge comme étant amie de cette classe.

8.2.2. Portée et limites

Avant de voir des opérateurs particuliers sur lesquels il convient d'attirer l'attention, voici quelques remarques d'ordre général concernant la définition de surcharges d'opérateurs.

a) Se limiter aux opérateurs existants

On ne peut surcharger que des opérateurs qui existent déjà pour des types de base. On ne donc pas créer de nouveaux symboles et il existe en outre des opérateurs qui ne peuvent pas être redéfinis. On est obligé de respecter le nombre d'opérandes de l'opérateur initial. Par exemple, l'opérateur + qui est un opérateur binaire (il admet deux opérandes) ne peut pas être surchargé pour une classe avec un unique argument. La surcharge des opérateurs ne change en rien les priorités relatives qu'ils ont avec les types de base, ni leur associativité. La liste des opérateurs que l'on peut surcharger est donnée dans l'annexe D.

b) Dépendance par rapport au contexte de la classe

On ne peut définir un opérateur que si celui-ci comporte au moins un argument d'un type d'une classe. La surcharge d'un opérateur est donc :

(1) une fonction membre : l'argument du type de la classe est implicite (c'est l'objet appelé),

(2) une fonction indépendante avec au moins 1 argument du type de la classe. La fonction est souvent amie de la classe.

Certains opérateurs sont forcément membres d'une classe : `=`, `()`, `[]`, `->`, `new` et `delete`.

Il faut aussi remarquer que le type de retour de la surcharge est important. Si l'opérateur que vous surchargez est un opérateur logique, la surcharge doit retourner

un objet du type `bool` (bien que le type `int` soit toléré). D'autres opérateurs bien spécifiques doivent retourner une valeur particulière : cela vous sera indiqué dans les sections suivantes.

c) Penser à la signification de l'opérateur

Il est important de remarquer que la surcharge d'un opérateur ne dépend que de votre bon sens. Vous pourriez très bien utiliser le symbole $\star$ pour faire la somme de deux nombres rationnels, par exemple. Pensez donc à utiliser des notations cohérentes avec celles en vigueur pour les types de base.

D'autre part, la surcharge de + et celle de = n'implique nullement le sens de += : ces trois opérateurs sont indépendants. Il convient donc de ne pas supposer le fonctionnement d'un opérateur, mais de le vérifier.

Le compilateur ne dispose que des règles de priorité relative et d'associativité des opérateurs.

8.2.3. Opérateurs ++ et - -

Un problème qui se pose lors de la surcharge des opérateurs ++ et - - est lié au fait qu'ils peuvent être utilisés en notation préfixée (par exemple, `++a`) ou en notation postfixée (par exemple, `a++`). Dans ces deux cas, les opérateurs ne sont pas les mêmes. Ainsi, `++a` n'utilise pas le même opérateur que `a++`. Il y a donc un problème d'identification de l'opérateur lors de la surcharge. Pour cela, un artifice a été créé en langage C++ puisque pour surcharger l'opérateur en notation postfixée, un argument fictif est utilisé :

- `type operator++()` (respectivement `type operator- -()`) est la surcharge de l'addition (respectivement soustraction) unitaire en notation *préfixée*.
- `type operator++(int)` (respectivement `type operator- -(int)`) est la surcharge de l'addition (respectivement soustraction) unitaire en notation *postfixée*.

Voici un exemple de surcharge, dans la classe `Cfraction`, des opérateurs ++ en notation préfixée et postfixée. Évidemment, la même chose peut être réalisée pour l'opérateur - -.

```
1  #include<stdio.h>
2
3  class Cfraction
4  {
5   public :
6           Cfraction &operator++()
7                   {
8                    printf("Opérateur préfixé\n");
```

```
9                   ...
10                  return *this;
11                 }
12          const Cfraction operator++(int iP1)
13                 {
14                  printf("Opérateur postfixé\n");
15                  ...
16                  return *this;
17                 }
18 };
19
20 void main()
21 {
22  Cfraction FRCfraction1;
23
24  FRCfraction1++;
25  ++FRCfraction1;
26 }
```

Faites maintenant attention aux types de retour des surcharges de l'exemple ci-dessus. La surcharge de l'opérateur en notation préfixée retourne l'objet modifié par référence : cela est logique puisque dans une expression, l'objet après exécution de la surcharge, peut être modifié sur la même ligne que celle où l'opérateur est appliqué. Par contre, la surcharge de l'opérateur en notation postfixée n'a pas besoin de retourner une référence puisque l'opérateur est appliqué en dernier dans une expression.

8.2.4. Opérateur d'affectation =

Tout comme le constructeur de recopie par défaut, l'opérateur d'affectation par défaut travaille membre par membre pour l'affectation d'objets d'une même classe. Donc, si lors de l'affectation d'un objet, celui-ci contient des membres qui sont des pointeurs, il convient de surcharger l'opérateur d'affectation. Dans ce cas, celui-ci est défini comme une fonction membre de la classe considérée, retournant une référence sur le type de sa classe d'appartenance. Ainsi la dernière ligne de la surcharge de l'opérateur est :

```
return *this;
```

Voici un exemple de surcharge :

```
1 #include<stdio.h>
2
3 class Cfraction
4 {
5  public :
6          Cfraction & operator=(Cfraction FRCP1)
```

```
7                 {
8                  printf("Surcharge de l'opérateur d'affectation\n");
9                  ...
10                 return *this;
11                }
12 };
13
14 void main()
15 {
16  Cfraction FRCfraction1,FRCfraction2;
17
18  FRCfraction1=FRCfraction2;
19 }
```

Cette surcharge de l'opérateur = permet de réaliser l'affectation d'un objet de type `Cfraction` dans un autre objet de type `Cfraction`.

La surcharge de l'opérateur d'affectation est toujours réalisée par une fonction membre d'une classe.
D'un point de vue génie logiciel vous veillerez bien à :

- Mettre systématiquement une surcharge du = lorsque vous surchargez le constructeur de recopie (voir la section 3.2.3. pour une explication sur le problème soulevé par une recopie membre à membre),
- Retourner une référence sur l'objet affecté en fin de surcharge.

Techniquement, le langage C++ vous autorise à retourner un objet qui ne soit pas une référence sur la classe. Par exemple, la surcharge suivante compile :

```
int operator=(Cfraction & FRCP1)
```

Il suffit juste que la surcharge de l'opérateur = renvoie un entier. *À quoi cela peut-il servir ?* Et bien tout simplement si vous réalisez des affectations au sein d'expressions plus complexes, comme par exemple :

```
int iV1=(FRCfraction1=FRCfraction2)+5;
```

À l'inverse, il est également possible de définir une surcharge de l'opérateur = qui prenne en argument un paramètre dont le type ne soit pas celui de la classe dans laquelle la surcharge est définie. En voici un exemple.

```
1 #include<stdio.h>
2
3 class Cfraction
```

```
4  {
5   public :
6           Cfraction & operator=(int iP1)
7                   {
8                    printf("Surcharge de l'opérateur d'affectation\n");
9                    ...
10                   return *this;
11                  }
12 };
13
14 void main()
15 {
16  Cfraction FRCfraction1;
17
18  FRCfraction1=5;
19 }
```

Dans cet exemple, on remarque donc que la ligne 18 provoque directement l'appel à la surcharge du = sans réaliser aucune conversion. Dans d'autres situations (si l'on n'avait pas surchargé l'opérateur = comme fait sur la ligne 6) la valeur 5 aurait dû être convertie, par exemple, dans le type `Cfraction`.

8.2.5. Opérateur d'indexation []

Il peut parfois être utile de surcharger l'opérateur [] plutôt que d'utiliser une fonction d'accès. Ainsi, si un objet `obj` contient un tableau d'entiers, il est plus naturel d'écrire `obj[i]` pour accéder à l'élément en position `i` dans le tableau plutôt que d'utiliser deux fonctions : une pour affecter une valeur, l'autre pour la lire. Il faut dans ce cas faire en sorte que la notation puisse être utilisée dans une expression (lecture, *rvalue*) et dans une affectation (écriture, *lvalue*). Pour cela, la valeur de retour fournie par l'opérateur doit être une référence.

> La surcharge de l'opérateur d'indexation est toujours réalisée par une fonction membre d'une classe.

Voici un exemple dans lequel nous avons une classe permettant de manipuler des listes d'objets de la classe `Cfraction`. La surcharge de l'opérateur [] permet de retourner l'élément se trouvant dans un position donnée dans la liste. Dans cet exemple, on suppose évidemment que la classe `Cfraction` soit définie avant la compilation de la classe `CListeFraction`.

```
1  class CListeFraction
2  {
3   private :
4           int iLFRtaille;
5           Cfraction LFRtableau[100];
6   public :
7            Cfraction & operator[](int iP1)
8                   {
9                    printf("Surcharge de l'opérateur d'indexation\n");
10                   return LFRtableau[iP1];
11                  }
12 };
13
14 void main()
15 {
16  CListeFraction LFRliste;
17  Cfraction FRCelement;
18
19  LFRliste[1]=FRCelement;
20 }
```

Dans cet exemple, l'objet retourné par la surcharge est passé par référence ce qui implique que sur la ligne 19 c'est l'objet d'origine qui se voit affecté le contenu de la variable `FRCelement`. Ainsi, *surcharger l'opérateur [] en lui faisant retourner une référence permet d'accéder en lecture et en modification aux éléments renvoyés*. Si vous souhaitez uniquement que cet opérateur soit utilisé en lecture, alors vous devez faire un retour par valeur ! Et oui, dans ce cas, ligne 19, c'est une copie du `Cfraction` en position 1 qui sera modifiée et pas le `Cfraction` se trouvant dans le tableau `LFRtableau`.

8.2.6. Opérateur d'accès aux membres d'une classe (->)

Il peut parfois être utile de surcharger le sélecteur de champs, c'est-à-dire l'opérateur `->`. Dans sa version standard, cet opérateur est utilisé sur un pointeur pour accéder aux éléments de l'objet stockés à l'adresse mémoire pointée. Cet opérateur peut être surchargé... mais pour les objets d'une classe. C'est-à-dire que sa surcharge est utilisée pour accéder aux éléments d'un objet et **non pas d'un pointeur**. Voici un exemple qui illustre cela.

```
1  #include<stdio.h>
2
3  class Cfraction
4  {
```

```
5   private :
6           int iFRCcompteur ;
7   public :
8           Cfraction() {iFRCcompteur=0 ;}
9
10          Cfraction * operator->()
11                  {
12                   printf("Surcharge de l'opérateur d'accès indirect\n");
13                   iFRCcompteur++ ;
14                   return this ;
15                  }
16
17          int FRCLire_Compteur() {return(iFRCcompteur) ;}
18  };
19
20  void main()
21  {
22   Cfraction *FRCfraction1 ;
23
24   FRCfraction1=new Cfraction ;
25   printf("Nombre d'accès : %d\n",FRCfraction1->FRCLire_Compteur()) ;
26  }
```

Dans cet exemple le nombre d'accès affiché est... 0. En effet, sur la ligne 25 l'opérateur -> est utilisé sur une adresse (`FRCfraction1`) et non pas sur un objet. Donc ce n'est pas la surcharge définie qui est appelée mais l'opérateur `->` standard. Par contre, l'exécution du code ci-dessous conduit à l'appel de la surcharge de l'opérateur `->`.

```
1   void main()
2   {
3    Cfraction FRCfraction1 ;
4
5    printf("Nombre d'accès : %d\n",FRCfraction1->FRCLire_Compteur()) ;
6   }
```

Il peut être intéressant de le surcharger, par exemple, si on souhaite effectuer un comptage des accès réalisés aux champs des objets. Cela peut être le cas également si votre classe contient un attribut du type pointeur sur un objet de la classe : l'opérateur `->` permet alors de retourner cet attribut. Ce sélecteur est une fonction membre qui renvoie un pointeur : l'adresse sur l'objet auquel on cherche à accéder. En quelque sorte, *la surcharge de cet opérateur vous permet de gérer un objet comme s'il était un pointeur sur un objet*.

La surcharge de l'opérateur d'accès est toujours réalisée par une fonction membre d'une classe.

Il peut être également intéressant, lorsqu'on surcharge l'opérateur `->` de surcharger l'opérateur de déréférencement noté `*` (à ne pas confondre avec l'opérateur de multiplication).

8.2.7. Opérateur de déréférencement (*)

Le langage C++ permet également de surcharger l'opérateur de déréferencement qui se note `*`. Attention, il ne faut pas confondre avec l'opérateur de multiplication, dont la surcharge prend nécessairement un argument de plus (lorsque tous les deux sont surchargés par une méthode). Comme pour l'opérateur d'accès `->`, l'opérateur de déréférencement permet de faire comme si un objet était un pointeur. Ainsi, on peut écrire `*FRCfraction1` alors que `FRCFraction1` est un objet de type `Cfraction` et non pas un pointeur. Voici au travers d'un exemple comment surcharger cet opérateur. Bien que celui-ci ne fasse pas grand-chose, vous remarquerez le type de l'objet de retour de la surcharge : il s'agit d'une référence, ce qui permet lors de l'utilisation de la surcharge de travailler directement sur l'objet d'origine et non pas une copie.

```
1  #include<stdio.h>
2
3  class Cfraction
4  {
5   public :
6           Cfraction & operator*()
7                   {
8                    printf("Surcharge de l'opérateur de déréférencement\n");
9                    return *this;
10                  }
11
12 };
13
14 void main()
15 {
16  Cfraction FRCfraction1;
17
18  *FRCfraction1;
19 }
```

8.2.8. Opérateurs `new` et `delete`

Il est possible de surcharger les opérateurs `new` et `delete` en ce qui concerne les classes que vous définissez. Cela peut être intéressant dans certains cas de figure : comptage de la mémoire allouée aux objets dynamiques d'une classe, gestion optimisée de l'allocation mémoire... Ce dernier point est très intéressant : si vous souhaitez optimiser l'allocation mémoire notamment en gérant un tas d'objets alloués mais non utilisés (parfois il est intéressant au point de vue optimisation d'allouer initialement un grand nombre d'objets et de s'en servir au cas par cas, ce qui évite des désallocations intempestives).

> Les surcharges des opérateurs `new` et `delete` sont toujours réalisées par des fonctions membres d'une classe.

Les impératifs à respecter pour la surcharge de ces deux opérateurs sont les suivants :

- `new` : la fonction membre suchargeant `new` doit recevoir un unique argument de type `size_t` et fournir en retour un pointeur de type `void *`.
- `delete` : la fonction membre surchargeant `delete` doit recevoir au minimum un unique argument du type `void *` (emplacement de l'objet à désallouer). Il ne doit retourner aucune valeur. Cette surcharge peut également recevoir un second paramètre du type `size_t` qui contient la taille de l'objet à désallouer. Remarquez que la valeur de ce paramètre est égale à cette du paramètre passé à la surcharge de l'opérateur `new`.

Voici un exemple qui illustre ce qui a été vu précédemment.

```
1  #include<stdio.h>
2
3  class Cfraction
4  {
5   public :
6           void * operator new(size_t taille)
7                   {
8                    void * pvV1 ;
9                    printf("Surcharge de l'opérateur new\n");
10                   pvV1= : :new char[taille] ;
11                   // pvV1 contient le bloc mémoire de taille octets alloué
12                   return pvV1 ;
13                  }
14          void operator delete(void *FRCP1)
15                  {
```

```
16                    printf("Surcharge de l'opérateur delete\n");
17                    ::delete FRCP1;
18                    // Le bloc mémoire est désalloué
19                   }
20 };
21
22 void main()
23 {
24  Cfraction *FRCfraction1;
25
26  //Allocation du pointeur
27  FRCfraction1=(Cfraction *)new Cfraction;
28
29  //Désallocation du pointeur
30  delete FRCfraction1;
31
32 }
```

Dans la surcharge de l'opérateur `new` (ligne 6), remarquez la syntaxe de la ligne 10 sur laquelle : `::new` représente un appel à l'opérateur `new` de base. Dans cet exemple, vous auriez pu omettre l'opérateur de résolution de portée car il n'y aurait eu aucun doute quant à la version de `new` que vous souhaitiez appeler. Néanmoins, cela ne sera peut être pas toujours le cas, auquel cas utilisez la syntaxe de la ligne 10. Un autre point intéressant concerne le type utilisé sur cette ligne : nous avons utilisé le type `char` dans le `new` puisque c'est ce type qui est d'une taille d'un octet, le paramètre `taille` indiquant le nombre d'octets à allouer.
La surcharge de l'opérateur `delete` (ligne 14) réalise ici simplement la désallocation de l'espace mémoire alloué. Notez que cette surcharge aurait pu prendre un second paramètre de type `size_t` qui aurait été la taille de la zone à désallouer.

Une question fondamentale se pose ici est : *Qui se charge de l'appel des constructeurs et des destructeurs ?* En effet, dans l'exemple précédent les surcharges que nous avons définies pour la classe `Cfraction` se comportent comme les fonctions `malloc` et `free` du langage C : aucun appel de constructeur ni de destructeur n'est précisé. Si vous réfléchissez bien, vous trouverez la réponse par vous même... Si votre surcharge de l'opérateur `new` ne possède aucune indication sur le constructeur à appeler (constructeur par défaut, par recopie, autre ?) alors c'est que ce n'est pas à elle d'appeler un constructeur. Et oui, le seul argument de votre surcharge est la taille de l'espace mémoire à allouer : elle n'a donc aucune information sur le constructeur à appeler. C'est donc le compilateur qui se charge de réaliser l'appel au bon constructeur juste après que votre surcharge ait été exécutée. Il en va de même pour la surcharge du destructeur.

Pour terminer, j'attire votre attention sur le point suivant.

Les opérateurs `new` et `delete` sont des fonctions membres statiques par défaut. Cela implique qu'elles n'accèdent qu'aux membres statiques de la classe dans laquelle elles sont surchargées. De plus, elles ne peuvent pas manipuler le pointeur d'auto-référencement (pointeur `this`).

Là encore, d'après vous quelle en est la raison ? Et bien tout simplement, la surcharge du `new` étant appelée avant la création de l'objet elle ne peut pas faire référence à des attributs qui ne sont pas encore alloués... De même, la surcharge de l'opérateur `delete` ayant en charge la désallocation de l'objet à tout moment, des membres autres que des membres statiques ne peuvent pas y être référencés.

8.3 LA SURCHARGE DE TYPES

La surcharge de types prend place dans le cadre des conversions entre objets et variables. Comme pour le langage C, il existe en langage C++ des conversions implicites et explicites. Ce dernier propose un mécanisme supplémentaire (la surcharge de types) qui permet à l'utilisateur de préciser ses propres règles de conversions entre des objets qu'il a défini. D'autre part, ces conversions sont très utiles car elles permettent d'éviter la surcharge de certains opérateurs. Cette section sur la surcharge de types va être l'occasion de revoir tous les mécanimes des conversions en distinguant ce qui se passe dans le cas des conversions implicites et des conversions explicites. Quatre types de conversions peuvent se produire :

(1) *Conversion d'un type de base (`int`, `float`...) vers un autre type de base*. Ce cas de figure ne sera pas plus abordé par la suite car il est du ressort du compilateur : vous ne pouvez pas définir vos propres conversions dans ce cas-là.

(2) *Conversion d'un objet d'une classe vers un type de base*.

(3) *Conversion d'un type de base vers un objet d'une classe*.

(4) *Conversion d'un objet d'une classe vers un objet d'une autre classe*.

Après un rappel sur les conversions ces trois derniers cas de figure seront vus plus en détail.

8.3.1. Rappels sur les conversions

Il existe deux types de conversions : les conversions explicites (appelées *cast*) et les conversions implicites. Une *conversion implicite* est réalisée de façon transparente par le compilateur sans que le programmeur n'intervienne.

Par exemple, le code suivant :

```
float fV1=5.3;
  int iV2=4;
float fV3=fV1 + iV2;
```

force le compilateur à convertir implicitement la variable `iV2` de `int` en `float` pour pouvoir réaliser l'addition.

Une conversion *explicite* fait appel à un opérateur de cast soit qui existe (entre types de base) soit qui est défini par l'utilisateur (conversion mettant en jeu au moins un objet d'une classe). Reprenons l'exemple ci-dessus mais en réalisant une conversion explicite :

```
float fV1=5.3;
  int iV2=4;
float fV3=(int)fV1 + iV2;
```

Le *cast* réalisé sur la dernière ligne utilise la syntaxe du langage C qui est dépréciée (mais toujours utilisable). En langage C++ vous auriez pu également écrire :

```
...
float fV3=int(fV1) + iV2;
```

Le langage C++ introduit également de nouveaux mots-clefs pour la conversion de variables/objets : `static_cast` et `dynamic_cast`. Ce premier est utilisé pour réaliser des conversions parfaitement définies au moment de la compilation tandis que le second est utilisé pour réaliser des conversions entre éléments n'étant complètement définis qu'au moment de l'exécution (c'est le cas notamment des classes contenant au moins une fonction virtuelle)[2]. Autrement dit, le mot-clef `static_cast` vient remplacer les deux syntaxes des conversions explicites rappelées ci-dessus. La syntaxe de `static_cast` et `dynamic_cast` est la suivante :

```
static_cast<type>(expression)
dynamic_cast<type>(expression)
```

L'exemple précédent devient donc :

```
float fV1=5.3;
  int iV2=4;
float fV3=static_cast<int>(fV1) + iV2;
```

D'un point de vue génie logiciel, l'utilisation des conversions explicites doit rester "raisonnable". En effet, si dans votre programme vous vous retrouvez avec trop de conversions explicites entre types de base alors cela veut dire que vous

2. L'opérateur `dynamic_cast` n'est également utilisable que pour convertir vers des pointeurs ou des références.

avez sans doute mal typé vos variables. De même, si vous avez trop de conversions explicites mettant en jeu des classes alors cela veut dire que vous n'avez sans doute pas créé assez de surcharges d'opérateurs pour que votre programme puisse fonctionner proprement. Dans ces deux cas, c'est à vous de remédier à la situation soit en changeant le type de certains variables soit en rajoutant les bonnes surcharges d'opérateurs.

Revenons maintenant sur la mise en place des conversions implicites. Dans une expression que doit réaliser le compilateur, lorsque l'opérateur utilisé n'existe pas pour les types des opérandes passées, le compilateur met en œuvre des chaînes de conversions implicites (une chaîne de conversions étant une série de changements de types pour un objet/variable donné).

Une chaîne de conversions est définie par au maximum 3 conversions dont au plus 1 conversion définie par l'utilisateur (CDU).

Ainsi, toute chaîne de conversions ne respectant pas la règle ci-dessus ne sera pas mise en œuvre par le compilateur (car trop de risques de perdre de l'information ou d'avoir au final un objet converti qui soit incohérent par rapport à l'objet de départ). Une conversion désigne ici nécessairement une *conversion implicite* qui peut être donc une des conversions listées au début de cette section. Une conversion est une *conversion définie par l'utilisateur* si elle ne met pas en jeu la conversion d'une variable d'un type de base vers un autre type de base. De plus il est important de noter que, par rapport à l'héritage, **l'utilisation d'un objet d'une classe fille à la place d'un objet d'une classe mère (par exemple, en paramètre d'une fonction, dans une expression...) est comptée comme une conversion implicite**. Cela raffine donc la règle qui avait été vue dans le chapitre 7, section 7.1.4., concernant la compatibilité entre la classe mère et la classe fille. Plus précisemment cela compte comme une conversion définie par l'utilisateur.

8.3.2. Conversion d'une classe vers un type de base

Il est possible en langage C++ de convertir un objet d'une classe dans un type de base. Pour cela, il suffit d'indiquer au compilateur ce qu'il faut faire en définissant au sein de la classe un opérateur de conversions, c'est-à-dire une surcharge du type de base vers lequel la conversion doit être réalisée. La syntaxe est identique à celle utilisée pour la surcharge des opérateurs. Par convention un opérateur de conversion retourne un objet du type de base de la surcharge, ce qui implique que le compilateur vous dispense (et même vous interdit !) de préciser un type de retour à la surcharge. La définition d'un opérateur de conversion est illustrée à l'aide de l'exemple suivant.

```
1  class Cfraction
2  {
3   private :
4           unsigned int uiNominateur ;
5           unsigned int uiDenominateur ;
6   public :
7            operator float() {return (float)uiNominateur/(float)uiDenominateur ;}
8  };
9
10 void main()
11 {
12  Cfraction FRCfraction1 ;
13  float fV1 ;
14
15  fV1=FRCfraction1 + 5.7 ;
16 }
```

Remarquez la syntaxe de la ligne 7 pour la surcharge du type `float` : aucun paramètre n'est passé et aucun objet n'est spécifié en retour. Vous pouvez surcharger n'importe quel type de base au sein d'une classe en utilisant la même syntaxe et en remplaçant le mot-clef `float` par le nom du type à surcharger. Dans l'exemple précédent, la surcharge du type `float` a rendu possible la compilation de la ligne 15. Le cheminement du compilateur est le suivant :

(1) Il n'existe pas de surcharge de l'opérateur + permettant de réaliser l'opération demandée, donc le compilateur va mettre en œuvre des chaînes de conversions : une pour l'objet `FRCfraction1` et une pour la valeur `5.7`.

(2) La question que se pose le compilateur est : *existe-t-il une chaîne de conversions par opérande qui me permette d'utiliser un opérateur + défini ?* Cela va donc passer par la conversion de `FRCfraction1` en une valeur réelle, l'addition entre deux réels étant définies de base dans le langage. Le compilateur appelle donc la surcharge que vous avez définie sur la ligne 7. Le résultat de l'addition est une valeur de type `float` qui est affectée dans la variable `fV1`. Ainsi est mise en œuvre la chaîne de conversion "`Cfraction` → `float`" qui contient 1 conversion dont 1 CDU.

8.3.3. Conversion d'un type de base vers une classe

La conversion d'une variable d'un type de base vers un objet d'une classe est réalisable en langage C++ via... la définition de constructeurs adéquats dans la classe concernée. Ainsi donc vous aviez déjà vu, mais sans le savoir, la conversion d'un type de base vers une classe puisque vous avez déjà défini des constructeurs. Comment

cela fonctionne-t-il dans le détail ? Reprenons l'exemple de la section précédente en le modifiant légèrement.

```
1  class Cfraction
2  {
3   private :
4           unsigned int uiNominateur ;
5           unsigned int uiDenominateur ;
6   public :
7            operator float() {return (float)uiNominateur/(float)uiDenominateur ;}
8            Cfraction() {}
9            Cfraction(float fP1) {uiNominateur=fP1*10000 ; uiDenominateur=10000 ;}
10 } ;
11
12 void main()
13 {
14  Cfraction FRCfraction1 ;
15
16  FRCfraction1=FRCfraction1 + 5.7 ;
17 }
```

La ligne 16 implique deux réalisations : tout d'abord évaluer l'expression `FRCfraction1+5.7` puis, en fonction du type du résultat, réaliser l'affectation dans `FRCfraction1`. L'expression est évaluée de la même façon que dans le premier exemple, c'est-à-dire en convertissant `FRCfraction1` en un `float` et en utilisant l'opérateur + défini pour ce type de base. Le résultat est donc un objet du type `float` qui doit être affecté dans un objet du type `Cfraction`.

(1) Pour réaliser ce deuxième travail, le compilateur va chercher s'il existe une surcharge de l'opérateur = dans la classe `Cfraction` qui prenne en argument un objet du type `float`. Comme cela n'est pas le cas, il va être nécessaire de mettre en place une chaîne de conversions.

(2) Comme nous sommes dans une affectation c'est l'objet à droite de l'opérateur d'affectation qui va être converti dans le type de l'objet de gauche, c'est-à-dire en `Cfraction`. Cette conversion est possible puisqu'au sein de cette classe il existe un constructeur à un argument prenant en argument un `float` (ligne 9). *Ainsi, un objet temporaire va être créé puis recopié membre à membre dans l'objet `FRCfraction1`.*

Ainsi, l'exemple compile et deux chaînes de conversions ont été mises en place : la première pour convertir l'objet `FRCfraction1` en `float` (1 conversion dont 1 CDU) et la seconde pour convertir un `float` en `Cfraction` (1 conversion dont 1 CDU).

Une possibilité vous est offerte par le langage C++ d'interdire l'utilisation d'un constructeur pour la conversion implicite d'un type de base vers une classe. Pour cela il vous suffit de déclarer le constructeur avec le mot-clef `explicit`.

```
explicit nom_constructeur(liste_parametres)
```

Par exemple, si dans l'exemple précédent la ligne 9 devient

```
explicit Cfraction(float fP1) ...
```

alors la ligne 16 n'est plus compilable : vous venez d'interdire la conversion *implicite* d'un `float` en un objet temporaire du type `Cfraction`. Par contre, vous pouvez toujours réaliser des conversions explicites.

Cette possibilité peut être intéressante lorsque vous déclarez ce constructeur pour pouvoir initialiser vos objets et pas pour réaliser des conversions. Ainsi, vous limiterez le risque d'introduire des ambiguïtés dans la réalisation de vos expressions (une ambiguïté survient lorsque plusieurs voies s'offrent au compilateur pour les réaliser).

8.3.4. Conversion d'une classe vers une autre classe

La conversion d'un objet d'une classe vers une autre classe est strictement identique à la conversion d'un type de base vers une classe, c'est-à-dire qu'elle passe par la définition d'un constructeur adéquate. Voici un exemple qui permet de convertir des objets du type `Creel` en objets du type `Cfraction`.

```
1  class Creel
2  {
3   ...
4  };
5
6  class Cfraction
7  {
8   public :
9              Cfraction() {}
10             Cfraction(Creel RELP1) {...}
11 };
12
13 void main()
14 {
15  Cfraction FRCfraction1;
16  Creel RELvaleur1;
17
18  FRCfraction1=RELvaleur1;
19 }
```

Là encore, vous pouvez déclarer le constructeur de la ligne 10 comme étant `explicit` ce qui n'autorisera que les conversions explicites. Dans ce cas la ligne 18 ne compilera pas.

8.3.5. Exemples de conversions

Dans cette section, plusieurs exemples de chaînes de conversions sont proposés et expliqués. Ils doivent vous permettre d'évaluer si vous avez compris comment fonctionnent les chaînes de conversions telles qu'elles ont été introduites dans ce chapitre.

```
1  class Cfraction
2  {
3   private :
4           unsigned int uiNominateur;
5           unsigned int uiDenominateur;
6   public :
7            // Déclaration des constructeurs
8            Cfraction() {}
9            Cfraction(float fP1) {uiNominateur=fP1*10000; uiDenominateur=10000;}
10           // Déclaration des surcharges d'opérateurs
11           Cfraction operator+(int iP1) {uiNominateur+=iP1*uiDenominateur; return *this;}
12           // Déclaration des surcharges de types
13           operator float() {return (float)uiNominateur/(float)uiDenominateur;}
14 };
15
16 void main()
17 {
18  Cfraction FRCfraction1;
19  Cfraction FRCfraction2;
20  float fV1;
21
22  FRCfraction1=FRCfraction1 + 5.7;
23  FRCfraction1=5.7 + FRCfraction1;
24  FRCfraction1=FRCfraction1 + 5;
25  fV1=FRCfraction1+5;
26  fV1=FRCfraction1+FRCfraction2;
27 }
```

Lisez tout d'abord l'interface de la classe `Cfraction` puis regardez les lignes 22 à 26. Vous devez retrouver les résultats suivants :

– Ligne 22 : cette ligne ne compile pas. Le message d'erreur qui s'affiche est "ambiguïté" puisque le compilateur dispose de plusieurs façons de réaliser l'expression. En effet, il peut soit convertir l'objet `FRCfraction1` en un `float` et réaliser l'addition, soit convertir la valeur 5.7 en un `int` et appeler la surcharge de l'opérateur + de la ligne 12.

– Ligne 23 : aussi surprenant que cela puisse paraître cette ligne compile. En effet, en inversant les deux opérandes (par rapport à la ligne 22) seule la conversion de l'objet `FRCfraction1` en un `float` permet de réaliser l'opération. Vous ne pouvez plus utiliser la surcharge de l'opérateur +. Il n'y a donc plus d'ambiguïté.
– Ligne 24 : cette ligne compile, bien qu'elle soit similaire à la ligne 22. En réalité, sur cette ligne le compilateur ne met en œuvre aucune chaîne de conversions puisqu'il trouve l'opérateur + (ligne 11) qui lui permet de réaliser exactement l'opération demandée.
– Ligne 25 : cette ligne compile. Tout comme la ligne 24, l'opération à droite du = est réalisable. L'affectation l'est également puisque le compilateur sait convertir un objet du type `Cfraction` (le résultat de l'addition) en un `float` puis en un `int`. Vous avez donc ici une chaîne à 2 conversions dont 1 CDU, ce qui est réalisable.
– Ligne 26 : cette ligne ne compile pas pour la même raison que pour la ligne 22, c'est-à-dire ambiguïté. Le compilateur n'arrive pas à réaliser l'opération `FRCfraction1 + FRCfraction2` car il dispose de plusieurs conversions possibles pour ces objets.

Ce qu'il est important de retenir c'est que le compilateur fonctionne toujours par étapes dans ses évaluations. Il prend toujours une opération qu'il chercher à réaliser. Pour cela, il va s'autoriser, si nécessaire, une chaîne de conversions par opérande. Puis, il va répéter le même processus avec le résultat de l'opération si celui-ci est concerné par une autre opération (c'est le cas si vous utilisez plusieurs opérations dans une même expression).

Chapitre 9

Les patrons de fonctions et de classes

9.1 DE LA GÉNÉRICITÉ AVEC LES PATRONS !

La généricité est un mécanisme qui va nous permettre d'écrire une seule fois un traitement ou une classe valable pour plusieurs types définis dynamiquement à l'utilisation. Le principe consiste à définir une fonction/méthode, ou une classe, avec non pas un (ou des) type(s), mais une (ou des) variable(s) représentant ces types. Prenons en exemple le cas d'une fonction/méthode : le langage C++ vous permet de définir une version d'une fonction qui prend un paramètre d'un type non défini à la compilation (par exemple, que vous allez nommer `typegenerique`). Ce type sera alors manipulé dans le code de la fonction comme s'il s'agissait d'un type connu. C'est simplement à chaque appel de cette fonction générique que le compilateur déterminera le type à considérer en fonction des paramètres passés.

La conception d'une fonction/méthode ou d'une classe générique se réalise souvent en 3 étapes. La première consiste a écrire partiellement, voir totalement, la fonction ou la classe avec un type et à assurer son fonctionnement au moins pour la construction et pour la destruction. Dans un second temps, on remplace chacun des types que l'on souhaite générique par un identifiant. On applique également une syntaxe particulière signifiant au compilateur que l'on souhaite une classe générique. Enfin, on utilise le code générique de la fonction, ou bien on instancie la classe générique en la dérivant.

Dans la suite de ce chapitre nous allons présenter les patrons de fonction, un patron de fonctions étant une fonction générique. Nous présenterons ensuite les patrons de classe.

9.2 LES PATRONS DE FONCTIONS/MÉTHODES

9.2.1. Création d'un patron de fonctions/méthodes

La création d'un patron se fait très simplement à l'aide du mot-clef `template` et en précisant le nom que vous donnez au type générique manipulé dans la fonction ou la méthode. Un patron peut être créé pour une fonction ou pour une méthode au sein d'une classe. Nous illustrons ci-dessous la création et la manipulation d'un patron de fonctions ; la syntaxe pour un patron de méthodes est similaire.

```
template <class nom_type_generique> type_retour nom_fonction(liste_parametres)
```

Autrement dit, on peut définir un patron de fonctions comme étant une fonction dont l'interface est précédée de `template <class nom_type_generique>`. Le mot-clef `template` indique au compilateur qu'il est en présence d'un patron tandis que le mot-clef `class` ne permet ici que de préciser le nom du type générique défini. Il ne faut pas confondre le mot-clef `class` ici avec celui qui est utilisé pour la déclaration des classes.

Notez bien qu'il faut nécessairement qu'un des paramètres du patron soit du type générique, sinon vous aurez une erreur de compilation. En effet, si cela n'était pas le cas, le compilateur ne pourrait pas déduire le type générique lors de l'appel au patron.

Il est également possible de déclarer un patron de fonctions/méthodes qui prenne plusieurs types génériques. La syntaxe est la suivante :

```
template <class nom_type_generique1, class nom_type_generique1, ...> type_retour
                        nom_fonction(liste_parametres)
```

Là encore il est nécessaire qu'il y ait au moins un paramètre de chaque type générique dans l'interface du patron.

9.2.2. Instanciation et utilisation

Avec l'interface présentée dans la section précédente, `nom_type_generique` peut être utilisé dans le patron de fonctions pour créer des variables automatiques ou pour typer des paramètres. Voici un exemple basique où un patron de fonctions nommé `Min3` est défini : celui-ci retourne la valeur la plus petite parmi les trois passées en paramètre.

```
1  #include<stdio.h>
2
3  template <class MType> MType Min3(MType MTPP1,MType MTPP2,MType MTPP3)
4  {
5   if (MTPP1 <= MTPP2 && MTPP1 <= MTPP3) return (MTPP1);
6   if (MTPP2 <= MTPP1 && MTPP2 <= MTPP3) return (MTPP2);
7   if (MTPP3 <= MTPP1 && MTPP3 <= MTPP2) return (MTPP3);
8  }
9
10 void main()
11 {
12  printf("Voici le min d'entiers : %d\n",Min3(5,6,7));
13  printf("Voici le min de réels  : %lf\n",Min3(5.1,6.2,7.3));
14 }
```

Cet exemple compile parfaitement et, à l'exécution, le type `MType` est remplacé par le type `int` sur la ligne 12 tandis que sur la ligne 13 il est remplacé par le type `double`.

> Lors de l'appel à une fonction/méthode pour laquelle il existe un patron, le compilateur cherche d'abord s'il existe une version compatible avec le type des paramètres passés lors de l'appel. S'il n'en existe pas, il crée à partir du patron un exemplaire de la fonction/méthode. Cet exemplaire n'est créé qu'une seule fois et valable dans tout le programme.

Regardez bien l'exemple ci-dessous : une version de la fonction `Min3` est créée qui ne soit pas un patron de fonctions.

```
1  int Min3(int iP1,int iP2, int iP3)
2  {
3   if (iP1 <= iP2 && iP1 <= iP3) return (iP1);
4   if (iP2 <= iP1 && iP2 <= iP3) return (iP2);
5   if (iP3 <= iP1 && iP3 <= iP2) return (iP3);
6  }
7
8  template <class MType> MType Min3(MType MTPP1,MType MTPP2,MType MTPP3)
9  {
10  if (MTPP1 <= MTPP2 && MTPP1 <= MTPP3) return (MTPP1);
11  if (MTPP2 <= MTPP1 && MTPP2 <= MTPP3) return (MTPP2);
12  if (MTPP3 <= MTPP1 && MTPP3 <= MTPP2) return (MTPP3);
13 }
14
```

```
15 void main()
16 {
17  printf("Voici le min d'entiers : %d\n",Min3(5,6,7));
18  printf("Voici le min de réels  : %lf\n",Min3(5.1,6.2,7.3));
19 }
```

Cette version prend en paramètre le type `int`. Ainsi, l'application de la règle encadrée ci-dessus montre que sur la ligne 17 c'est la version de la ligne 1 qui est appelée : il n'y a pas de fonction de créée à partir du patron défini sur la ligne 8. Par contre, la ligne 18 provoque la création d'un exemplaire de la fonction `Min3` à partir du patron.

Voici maintenant une autre utilisation du patron de fonctions `Min3`, plus déroutante.

```
1  #include<stdio.h>
2
3  template <class MType> MType Min3(MType MTPP1,MType MTPP2,MType MTPP3)
4  {
5   if (MTPP1 <= MTPP2 && MTPP1 <= MTPP3) return (MTPP1);
6   if (MTPP2 <= MTPP1 && MTPP2 <= MTPP3) return (MTPP2);
7   if (MTPP3 <= MTPP1 && MTPP3 <= MTPP2) return (MTPP3);
8  }
9
10 class Cfraction
11 {
12  ...
13 };
14
15 void main()
16 {
17  Cfraction FRCV1,FRCV2,FRCV3;
18  printf("Voici le min de trois fractions : %d\n",Min3(FRCV1,FRCV2,FRCV3));
19 }
```

Cette fois-ci la fonction `main` ne compile plus et vous avez le message d'erreur : *opérateur binaire "`>=`" non défini pour la classe `Cfraction`*. Ce résultat est tout à fait logique puisque, sur la ligne 18, le type `MType` est remplacé par le type `Cfraction`. Autrement dit, sur les lignes 5 à 7 c'est la comparaison entre des objets de la classe `Cfraction` qui est réalisée. Or, pour que cela compile il faut que l'opérateur "<=" soit défini pour cette classe (ou alors qu'il existe des chaînes de conversion permettant de se ramener à une version existante de cet opérateur).

Vous avez également la possibilité en langage C++ de préciser l'instanciation du patron de fonctions que vous voulez appeler. Ainsi, même si vous appeler un patron de fonctions en lui passant le type `int` comme type générique, vous pouvez appeler

la version du patron qui travaille avec le type `double`. Pour cela, il suffit juste de le préciser lors de l'appel au patron comme illustré dans l'exemple ci-dessous.

```
1 ...
2 void main()
3 {
4  printf("Voici le min de trois entiers : %d\n",Min3<double>(5,7,2));
5 }
```

Dans cet exemple, c'est l'instanciation du patron prenant le type `double` en type générique qui sera appelée, bien que ce soient des arguments de type `int` qui soient passés.

9.2.3. Spécialisation d'un patron de fonctions/méthodes

Il est possible de spécialiser un patron de fonctions/méthodes, c'est-à-dire de préciser pour un type particulier le code à exécuter. La syntaxe d'une spécialisation est illustrée dans l'exemple suivant.

```
1  template <class MType> MType Min3(MType MTPP1,MType MTPP2,MType MTPP3)
2  { // Le patron de classe
3   if (MTPP1 <= MTPP2 && MTPP1 <= MTPP3) return (MTPP1);
4   if (MTPP2 <= MTPP1 && MTPP2 <= MTPP3) return (MTPP2);
5   if (MTPP3 <= MTPP1 && MTPP3 <= MTPP2) return (MTPP3);
6  }
7
8  template <> int Min3<int>(int iP1,int iP2,int iP3)
9  { // Sa spécialisation au type int
10  if (iP1 <= iP2 && iP1 <= iP3) return (iP1);
11  if (iP2 <= iP1 && iP2 <= iP3) return (iP2);
12  if (iP3 <= iP1 && iP3 <= iP2) return (iP3);
13 }
```

Dans cet exemple, la spécialisation contient le même code que le patron, mais cela n'est pas une obligation. Vous remarquerez que le nom du type générique n'est pas précisé : seul le type `int` est mentionné après le nom de la spécialisation. *Une spécialisation suit nécessairement la définition du patron.*

9.2.4. Sucharger un patron de fonctions/méthodes

Il y a plusieurs façons de surcharger un patron de fonctions/méthodes : on peut le surcharger soit par une fonction/méthode, soit par un autre patron. Nous avons déjà rencontré le premier cas de figure dans la section 9.2.2. lorsque nous avons défini la version suivante de la fonnction `Min3` :

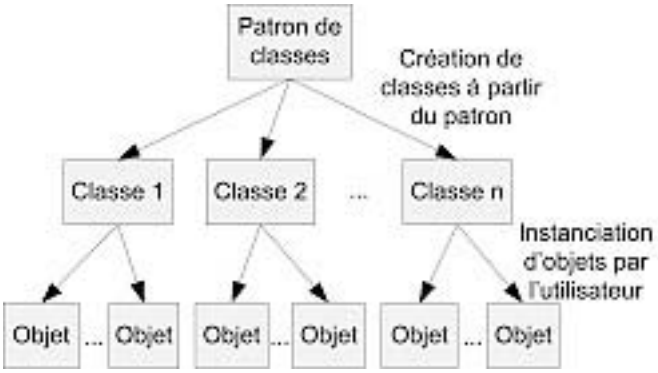


FIG. 9.1: Hiérarchie des patrons, classes et objets

```
int Min3(int iP1, int iP2, int iP3).
```

Comme nous l'avons déjà souligné, lors de l'appel cette version surchargée sera systématiquement privilégiée si les paramètres passés sont de type `int` et aucune instanciation du patron ne sera faite.

Il est également possible de surcharger un patron par un autre patron. Dans ce cas vous devez simplement définir à la suite les deux patrons, et il n'y a aucune syntaxe particulière. À l'appel d'un patron, le compilateur choisira en fonction du type des paramètres passés. Notez bien que le fait de définir plusieurs patrons ne peut que rajouter des risques d'erreurs de compilation du type "ambiguïté" puisqu'un problème de choix se pose au compilateur.

9.3 LES PATRONS DE CLASSES

On peut appliquer le même mécanisme de généricité pour les classes. Afin d'obtenir une version utilisable d'une classe générique, il faudra instancier la classe générique soit en la dérivant soit en déclarant un objet de cette classe en précisant les types génériques (comme ce que vous avez vu pour les patrons de fonction/méthode). Même si le principe ressemble à celui appliqué aux fonctions, d'importantes différences existent entre les patrons de fonctions/méthodes et les patrons de classes.

Un patron de classes peut être vu comme un modèle pour la création dynamique de classes similaires. La figure 9.1 montre la place d'un patron de classes par rapport aux classes et aux objets : à partir d'un patron vous pouvez instancier plusieurs classes et chaque classe vous permet de créer plusieurs objets que vos programmes vont manipuler.

9.3.1. Création d'un patron de classes

La création d'un patron de classes se fait selon une syntaxe proche de celle utilisée pour les patrons de fonction/méthode. Nous l'illustrons sur un exemple.

```
1  template <class MType> class Cliste
2  {
3   private :
4    int iTaille ;
5    MType * pMTPListe ;
6    // Éléments permettant de définir le contenu de la liste
7   public :
8            Cliste<MType>() ; // Constructeur par défaut
9            Cliste<MType>(Cliste<MType> & MTPP1) ; // Constructeur de recopie
10           int LISLire_taille() ;
11           ...
12 };
```

Remarquez tout d'abord la syntaxe de la ligne 1 qui permet de déclarer la classe : l'utilisation des mots-clefs `template` et `<class >` permet de spécifier (comme pour les patrons de fonction/méthode) que nous sommes en train de déclarer un patron de classes. À partir de là, le nom du patron dans la suite est nécessairement `Cliste<MType>` et non pas simplement `Cliste`. C'est pour cela que les lignes 8 et 9 font mention du type `MType` dans la déclaration des constructeurs. *Vous constatez donc bien que la syntaxe d'un patron de classes est relativement lourde par rapport à une classe simple*.

Bien évidemment vous pouvez, comme pour les patrons de fonction/méthode, déclarer plusieurs types génériques au sein du patron de classes. Il suffit pour cela de faire figurer plusieurs fois le mot-clef `class` entre chevrons sur la ligne 1. Là encore le nom de la classe est à modifier dans les lignes 8 et 9 en conséquence.

La définition des méthodes du patron de classes suit la même logique que la définition d'une méthode d'une classe au détail près que le nom du patron est plus long que le nom d'une classe puisqu'on doit faire figurer la mention des types génériques. Vous pouvez toujours définir une méthode dans la déclaration de la classe, auquel cas la méthode sera `inline`. Vous pouvez aussi la définir en dehors de la déclaration de la classe **mais dans ce cas vous devrez la définir à la suite de cette déclaration dans le fichier `.h` !** Sur l'exemple précédent, la définition du constructeur de recopie est la suivante.

```
1  template <class MType> class Cliste
2  {
3   private :
4    int iTaille ;
5    MType * pMTPListe ;
```

```
6    // Élements permettant de définir le contenu de la liste
7   public :
8           Cliste<MType>(); // Constructeur par défaut
9           Cliste<MType>(Cliste<MType> & MTPP1); // Constructeur de recopie
10          int LISLire_taille();
11          ...
12  };
13
14  template <class MType> int Cliste<MType> : :Cliste<MType>(Cliste<MType> &MTPP1)
15  {
16          ...
17  }
```

Cette définition vous montre bien qu'une méthode d'un patron de classes est en réalité un patron de méthodes. Pour définir cette méthode `inline`, et puisqu'elle est définie dans le fichier `.cpp`, il vous suffit de faire figurer le mot-clef `inline` après la spécification du template de méthode, *i.e.* "`template <class MType>`".

Revenons un instant sur *la définition des méthodes au sein du fichier `.h`*. La déclaration d'un patron de classes vous y oblige par défaut. Certes ! Néanmoins, cela a l'inconvénient que le code est compilé à chaque fois que l'interface de votre patron de classes est inclu dans un autre fichier. Pour remédier à cela, vous avez la possiblité d'exporter explicitement la définition de vos méthodes dans un fichier `.cpp`. Ainsi vous vous ramenez à une programmation propre puisque l'interface de votre classe va se retrouver séparée de son corps. Pour réaliser cette exportation il suffit, lors de la définition des méthodes (dans le fichier `.cpp`) de les faire précéder du mot-clef `export`.

```
export template <class type1,...>
nom_patron : :nom_methode(parametres)
```

Si vous utilisez le mot-clef `export` lors de la déclaration du patron de classes (juste avant le mot-clef `template`) vous provoquerez l'exportation automatique de la définition de toutes les méthodes qui ne le sont pas dans la déclaration du patron de classes.

> Autant que possible vous essayerez d'exporter la définition des méthodes de vos patrons de classe dans le fichier `.cpp` associé de sorte à préserver la délimitation entre l'interface et le corps.

9.3.2. Les membres statiques au sein d'un patron

Il est possible de déclarer au sein d'un patron de classes des membres statiques. Je rappelle ici qu'un attribut statique d'une classe est un attribut qui est indépendant des

objets de la classe : tous ont accès à ce même attribut. Dans le cas d'une méthode statique au sein d'une classe, le raisonnement est le même : une méthode statique peut être appelée indépendamment des objets de la classe et ne peut référencer que des attributs statiques.

Dans le cas des patrons de classes, une méthode statique du patron sera considérée comme une méthode statique de chaque classe créée à partir du patron. Ainsi, cette méthode accèdera à tous les attributs statiques *de la classe créée à partir du patron*.

En ce qui concerne les attributs statiques, la situation est un peu différente dans le cas des patrons de classe.

9.3.3. Instanciation et utilisation

Il existe deux types d'instanciation : l'instanciation implicite et l'instanciation explicite. La première consiste à créer la classe, à partir du patron, lors de la déclaration d'un objet de la classe. Voici un exemple *d'instanciation implicite* qui reprend le patron `Cliste` défini dans la section précédente.

```
1 void main()
2 {
3  Cliste<int> LISliste_entiers;
4  // Une classe est créée via la déclaration de l'objet LISliste_entiers
5  ...
6 }
```

La déclaration de l'objet `LISliste_entiers` sur la ligne 3 provoque plusieurs traitements : (i) si la classe `Cliste<int>` n'avait pas été créée auparavant à partir du patron `Cliste`, alors elle l'est, (ii) l'objet est déclaré et initialisé par appel au constructeur approprié. Notez bien que techniquement, les méthodes de votre classe ne sont générées que lorsque vous les utilisez. Ainsi, si vous avec un patron de méthodes dans votre patron de classes qui n'est jamais appelé alors aucune méthode ne sera générée à partir du patron de méthodes.

Il est important de bien faire attention à ce que vous déclarez. Par exemple, le code suivant ne provoque pas l'instanciation d'une classe à partir du patron puisque vous ne déclarez qu'un pointeur sur une classe et non pas un objet.

```
1 void main()
2 {
3  Cliste<int> * pLISliste_entiers;
4  // Un pointeur sur une liste d'entiers est déclaré
5  ...
6 }
```

Comme je l'ai suggéré précédemment il est possible de déclarer explicitement des classes à partir de leur patron. Pour cela, il suffit d'utiliser la syntaxe suivante.

```
template class nom_patron<type1,type2,...>
```

```
1 void main()
2 {
3  template class Cliste<float>;
4  // Instanciation explicite de la classe Cliste
5  ...
6 }
```

L'exemple précédent illustre la syntaxe d'instanciation explicite d'une classe à partir de son patron. Ainsi, la ligne 3 va provoquer la création de la classe `Cliste<float>` et *l'instanciation de toutes ses méthodes à partir de leur patron de méthodes*. Il s'agit bien évidemment d'une technique du langage C++ que vous serez amener à peu manipuler en pratique.

9.3.4. Préciser des types par défaut pour les types génériques

Lors de la déclaration d'un patron de classes, vous avez la possibilité de préciser des types par défaut pour certains types génériques. Sur le principe, il s'agit d'un mécanisme fortement similaire à celui permettant de définir des valeurs par défaut aux arguments d'une fonction (cf. section 4.1).

Pour déclarer des types par défaut, il suffit lors de la déclaration des types génériques de préciser ceux qui possèdent un type par défaut et lequel.

```
template <class T1, class T2=int, ..., class Tn=float> class
                        nom_classe {...
```

Vous remarquerez ici que, comme dans le cas d'arguments passés à une fonction, les types par défaut sont nécessairement mis de la droite vers la gauche : à partir du moment où un type générique `Ti` possède un type par défaut alors tous les types génériques qui suivent doivent en avoir un. Cela est nécessaire pour que le compilateur sache, à l'instanciation des classes, où mettre les types que vous passez en paramètre. Voici un exemple qui éclaircira sans doute la situation.

```
1  template <class MType1, class MType2=int> class CDoubleListe
2  { // Cette classe permet de gérer une double liste d'éléments
3    // de types hétérogènes
4   private :
5    int iTaille ;
6    MType1 * pMTPListe1 ;
7    MType2 * pmTPListe2 ;
8    // Élements permettant de définir le contenu des listes
9   public :
10   CDoubleListe<MType1,MType2>() ; // Constructeur par défaut
11   CDoubleListe<MType1,MType2>(CDoubleListe<MType1,MType2> & MTPP1) ;
12    // Constructeur de recopie
13 ...
14 } ;
15 ...
16 void main ()
17 {
18  CDoubleListe<float> DLIV1 ; // Double liste de type (float,int)
19  CDoubleListe<float,float> DLIV2 ; // Double liste de type (float,float)
20  ...
21 }
```

Remarquez la syntaxe des lignes 1 et 19. Comme ligne 1 vous avez mis une valeur par défaut pour le type générique `MType2`, vous avez pu ligne 19 déclarer un objet d'une classe n'instanciant qu'un type générique sur les deux.

9.3.5. L'amitié et les patrons

Comme pour les classes, il est possible de déclarer que des fonctions, méthodes, classes ou patrons peuvent être *amis* d'un patron de classes. Nous distinguons trois cas de figure possibles :

(1) *Vous voulez déclarer qu'une fonction, une méthode ou une classe est amie d'un patron de classes.* Si une fonction ou une méthode est amie d'un patron de classes alors elle sera amie de toutes les classes instanciées à partir du patron de classes. De même, dans le cas d'une classe `A` amie d'un patron, toutes les méthodes de la classe `A` seront amies de toutes les classes obtenues par instanciation du patron. D'un point de vue syntaxique, déclarer qu'une fonction, méthode ou classe est amie d'un patron est strictement similaire à la déclaration d'amitié dans le cas d'une classe (cf. chapitre 5).

(2) *Vous voulez déclarer qu'un patron de fonctions est ami d'un patron de classes.* Dans ce cas, seule une instance du patron de fonctions sera amie d'une instance du patron de classes. Autrement dit, dès que vous instancierez une classe à partir

de son patron, vous provoquerez l'instanciation d'une fonction amie à partir du patron de fonctions. Voici, un exemple de déclaration de patron de fonctions ami d'un patron de classes.

```
1  template <class MType> class Cliste
2  {
3   private :
4    int iTaille ;
5    MType * pMTPListe ;
6
7   public :
8    friend Cliste<MType> CLCminimum() ;
9          // Cette fonction cherche l'élément minimum au sein de la liste
10 ...
11 };
12 ...
13 void main ()
14 {
15  Cliste<float> LISV1 ; // Cette déclaration d'objet provoque la création
16                        // d'une classe à partir du patron et la création
17                        // d' une méthode CLCminimum à partir de son patron.
18  ...
19 }
```

(3) *Vous voulez déclarer qu'un patron de classes ou un patron de fonctions est ami d'une classe*. Dans ce cas, toutes les instances des patrons sont amis de la classe. La syntaxe est la même que pour la déclaration d'amitié classique, à la nuance près que le nom des patrons est toujours accompagné du nom des types génériques entre chevrons.

9.3.6. Spécialisation du patron

Il peut arriver des situations où vous souhaitez particulariser le comportement d'un patron de classes dans le cas où le type générique possède certaines valeurs. Il vous est possible en langage C++ de spécialiser le patron de classes dans ce cas. Ainsi, une spécialisation est bien plus que la particularisation du patron à un type : la spécialisation peut contenir des traitements et attributs particuliers que ne contient pas le patron.

Reprenons l'exemple de la classe `Cliste` pour illustrer la syntaxe de la spécialisation d'un patron de classes.

```
1  template <class MType> class Cliste
2  { // Déclaration du patron de classe
3   ...
```

```
4 };
5
6 template <> class Liste<char *>
7 { // Déclaration de la spécialisation du patron
8   // de classe pour le type char *
9 ...
10 };
11 ...
12 void main ()
13 {
14  Liste<float> LISV1; // Fait référence au patron
15  Liste<char *> LISV2; // Fait référence à la spécialisation du patron
16  ...
17 }
```

Comparez bien la syntaxe des lignes 1 et 6. Sur la ligne 6 aucun type générique n'est précisé après le mot-clef `template` tandis qu'un type est précisé après le nom du patron. C'est ainsi que nous précisons une spécialisation. De plus, notez bien que celle-ci doit venir après la déclaration de la classe, "après" pouvant simplement vouloir dire que le fichier contenant la spécialisation inclut le fichier contenant la déclaration du patron.

Dans cet exemple la **spécialisation est totale**, or le langage C++ vous permet également de faire de la **spécialisation partielle** dans le cas d'un patron de classes ayant plusieurs types génériques. Cette seconde spécialisation consiste à laisser la déclaration de certains types génériques tandis que d'autres sont définis comme sur la ligne 6 ci-dessus. Voici un exemple de spécialisation partielle pour la classe `CDoubleListe`.

```
1 template <class MType1> class CDoubleListe<int,MType1>
2 { // Spécialisation partielle du patron de classe
3 ...
4 };
```

Notez bien que lors de la spécialisation partielle le nom du patron est suivi, entre chevrons, du même nombre d'arguments qu'il a été déclaré de type générique dans le patron de base.

9.3.7. Aller encore plus loin avec les patrons

Les patrons de classes offrent de nombreuses possibilités à l'utilisateur pour qu'il puisse introduire de la généricité dans ses programmes. Les patrons de classes permettent bien plus que ce qui est présenté dans ce chapitre et si vous êtes intéressés vous pouvez consulter la référence [**3**] pour de plus amples informations d'ordre techniques. Je tiens juste à attirer votre attention sur le fait que l'inconvénient des patrons

de classe est que très vite la lecture et l'écriture d'un programme peuvent devenir fastidieuses. En conséquent, plutôt que de favoriser la technicité de votre programme pensez plutôt à en facilité sa lecture et sa maintenance. Ainsi, l'utilisation des patrons de classe doit rester parcimonieuse et mesurée.

Chapitre **10**

Introduction au polymorphisme et aux méthodes virtuelles

10.1 QU'EST-CE QUE LE POLYMORPHISME ?

Polymorphisme est un terme qui désigne la capacité d'un "objet" à prendre plusieurs formes au cours de sa vie. Il ne faut pas confondre *polymorphisme* et *généricité* : la généricité permet de créer plusieurs classes ou objets à partir d'un cadre commun, c'est ce que nous avons vu au chapitre 9 avec les patrons. Le polymorphisme permet à un même "objet" de changer de forme au cours de sa vie. En réalité, en langage C++, **ce n'est pas un objet mais un pointeur sur un objet qui peut "changer de forme"**. Pour poursuivre cette introduction au polymorphisme, considérez l'exemple suivant.

```
1  #include<stdio.h>
2
3  class Cmere
4  {
5   public :
6            void Afficher() {printf("Objet de la classe Cmere\n");}
7  };
8
9  class Cfille : public Cmere
10 {
11  public :
```

```
12          void Afficher() {printf("Objet de la classe Cfille\n");}
13 };
14
15 void main()
16 {
17  Cmere * pMERV1;
18
19  Cmere MERV2;
20  Cfille FILV3;
21
22  pMERV1 = & MERV2; // pMERV1 pointe maintenant sur l'objet MERV2
23  pMERV1->Afficher();
24  pMERV1 = & FILV3; // pMERV1 pointe maintenant sur l'objet FILV3
25  pMERV1->Afficher();
26 }
```

L'affichage résultant est le suivant :

```
Objet de la classe Cmere
Objet de la classe Cmere
```

Revenons sur la ligne 25 dont on aurait pu supposer, à première vue, qu'elle provoque l'appel de la méthode de la ligne 12. Le pointeur `pMERV1` est un pointeur sur un objet de la classe `Cmere`. L'objet `FILV3` est un objet de la classe `Cfille`, ce qui implique qu'il possède les attributs et les méthodes de sa classe mère, la classe `Cmere`. Or "`pMERV1->`" fait référence à *l'objet de la classe `Cmere` pointé*, et ainsi `pMERV1->Afficher()` fait bien référence à la méthode de la ligne 6. L'affichage est donc tout à fait logique !

La question est donc : **comment faire pour que sur la ligne 23 ce soit la méthode de la classe `Cmere` qui soit utilisée et celle de la classe `Cfille` sur la ligne 25 ?** Evidemment, la contrainte est de toujours utiliser pour cela le pointeur `pMERV1` sur la classe `Cmere`... et bien c'est justement ici qu'intervient le polymorphisme : avec un même pointeur appeler différentes méthodes en fonction du type réel de l'objet pointé.

Le mécanisme du polymorphisme en langage C++ est donc défini dans un cadre très précis : vous avez une hiérarchie de classes liées par héritage et un pointeur sur une classe de la hiérarchie. Le polymorphisme vous permet alors de déterminer la bonne version d'une méthode en fonction de l'objet pointé.

Le polymorphisme est rendu possible grâce à ce que l'on appelle *le linkage dynamique*. Lorsque, dans un programme, une fonction `f` appelle une fonction `g` alors à l'étape du linkage le compilateur réalise l'appel effectif de la fonction `g` au sein de la fonction `f` en utilisant son "adresse". Il s'agit ici du linkage statique car la fonction `g` est bien identifiée lors de la compilation. Le linkage dynamique est un peu différent puisqu'en l'occurence dans mon exemple, il implique que l'appel effectif de la

fonction `g` n'est déterminé que lors de l'exécution de la fonction `f`, en temps réel en quelque sorte : lorsque `f` s'exécute elle détermine l'adresse de la fonction `g` à appeler. Vous comprenez donc que, mécaniquement, le polymorphisme est une simple application du linkage dynamique.

Dans la suite de ce chapitre nous allons voir comment, d'un point de vue syntaxique, mettre en œuvre le polymorphisme. Pour cela, nous allons voir la notion de méthode virtuelle.

10.2 LES MÉTHODES VIRTUELLES

Une méthode virtuelle est une méthode polymorphique, forcément définie plusieurs fois au sein d'une hiérarchie de classes liées par héritage.

10.2.1. Définition et déclaration

Pour déclarer qu'une méthode est virtuelle, il suffit de faire précéder sa déclaration du mot-clef `virtual`. Voici l'exemple, modifié, des classes `Cmere` et `Cfille` de la section précédente.

```
1  #include<stdio.h>
2
3  class Cmere
4  {
5   public :
6            virtual void Afficher() {printf("Objet de la classe Cmere\n");}
7  };
8
9  class Cfille : public Cmere
10 {
11  public :
12           virtual void Afficher() {printf("Objet de la classe Cfille\n");}
13 };
14
15 void main()
16 {
17  Cmere * pMERV1;
18  Cmere MERV2;
19  Cfille FILV3;
20  pMERV1 = & MERV2; // pMERV1 pointe maintenant sur l'objet MERV2
21  pMERV1->Afficher();
22  pMERV1 = & FILV3; // pMERV1 pointe maintenant sur l'objet FILV3
23  pMERV1->Afficher();
24 }
```

L'affichage résultant est le suivant...

```
Objet de la classe Cmere
Objet de la classe Cfille
```

... ce qui vous montre bien l'aspect polymorphique du pointeur `pMERV1` et surtout de la méthode `Afficher`. *C'est le type de l'objet pointé qui détermine la version de la méthode virtuelle à appeler*. Par contre, il faut bien que vous ayez en tête que l'appel :

```
FILV3->Afficher();
```

provoque **toujours** l'appel à la méthode définie sur la ligne 12 et qu'il ne met pas en jeu le polymorphisme. Dans cet appel, c'est du linkage statique qui est utilisé par le compilateur car vous ne passez pas par un pointeur.

Notez que si vous souhaitez quand même appeler, sur la ligne 25, la version de `Afficher` de la classe `Cmere`, vous devez utiliser l'opérateur de résolution de porter pour contourner le polymorphisme : `pMERV1->Cmere::Afficher();`.

Voici maintenant quelques règles à respecter lorsque vous créez des fonctions virtuelles.

(1) La présence du mot-clef `virtual` est nécessaire lors de la déclaration de la méthode (fichier `.h`). Elle est interdite lors de sa définition (fichier `.cpp`) : vous obtiendrez dans ce cas un message d'erreur à la compilation.

(2) La présence du mot-clef `virtual` est facultative lors de la déclaration de la méthode virtuelle dans les classes filles. Ainsi, dans l'exemple précédent, sur la ligne 12 le mot-clef `virtual` est optionnel puisque la méthode a été déclarée comme telle dans la classe mère.

(3) Une méthode que vous déclarez virtuelle l'est forcément, de fait, dans toutes les classes filles. Elle doit également être déclarée et définie avec les mêmes paramètres (sinon vous ne pourrez pas faire marcher le polymorphisme) dans toutes les classes filles[1]. Si dans la classe mère la méthode virtuelle est définie comme une méthode constante alors elle doit également l'être dans les classes filles.

> D'un point de vue génie logiciel, et pour des questions de lisibilité du code, je vous conseille de systématiquement faire figurer le mot-clef `virtual` dans les classes filles pour les méthodes virtuelles.

1. Cela vaut pour les paramètres d'entrée. L'objet de retour d'une méthode virtuelle peut différer légèrement d'une classe mère à une classe fille si, au sein de la classe mère, la méthode retourne un pointeur ou une référence sur une classe `A`. Dans ce cas, les versions de la méthode des classes filles peuvent retourner une adresse ou une référence sur une classe dérivée de la classe `A`.

Précisons également que si une méthode virtuelle est appelée au sein d'une autre méthode de la classe, nommée `f` par exemple, alors l'appel à la méthode `f` via un pointeur sur une classe mère provoque quand même l'appel de la méthode virtuelle définie au sein de la classe (et non pas la version de la classe mère, bien que la version de `f` appelée soit celle de la classe mère).

> Dans une classe contenant une fonction virtuelle et des attributs dynamiques, je vous conseille fortement de déclarer le destructeur comme étant virtuel également. Cela implique qu'à chaque destruction d'objets d'une classe fille via un pointeur d'une classe mère, c'est bien le destructeur de la classe fille qui fera les désallocations des attributs dynamiques. Vous éviterez ainsi d'avoir des fuites mémoires dues à l'appel d'un destructeur de la classe mère ne désallouant pas les attributs dynamiques de la classe fille.

10.2.2. Méthodes virtuelles et héritages complexes

Je tiens à attirer votre attention sur le fait que la gestion des méthodes virtuelles peut vite s'avérer fastidieuse lorsque nous sommes en présence d'une hiérarchie de classes complexes. Pour illustrer ce propos, je vous renvoie à la figure 10.1 qui donne un exemple de hiérarchie de classes un peu complexe. Dans cette figure, les liens d'héritage sont indiqués : les classes 3 et 4 héritent de la classe 1 et la classe 5 hérite des classes 2 et 4. Par ailleurs, je suppose dans cet exemple que lorsque pour une classe figure la mention `virtual X` alors la méthode `X` est définie comme une méthode virtuelle au sein de la classe.

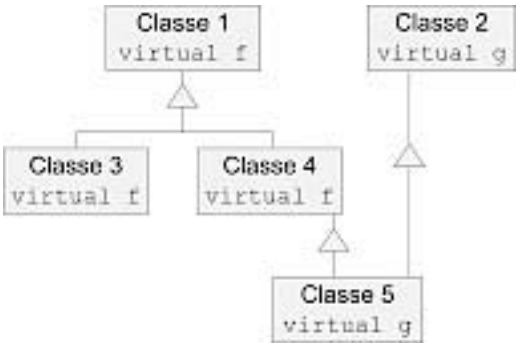


FIG. 10.1: Un exemple d'héritage complexe

La première remarque qui vient est que la classe 5 ne redéfinit pas la méthode virtuelle `f` qui lui vient de la classe 4. Que se passe-t-il dans ce cas-là ? Et bien tout

simplement, la classe 5 hérite de la version de la méthode `f` définie dans la classe 4. Supposez maintenant que le code suivant soit exécuté :

```
pC1V1= & C5V2 ;
  pC1V1->f() ;
```

avec `pC1V1` un pointeur sur la classe 1 et `C5V2` un objet de la classe 5. *Quelle version de la méthode `f` est exécutée ?* Et bien, la version héritée de la classe 4. La méthode `f` étant polymorphique, **le linkage dynamique retiendra la version de `f` la plus proche de la classe 5 puisque `pC1V1` pointe sur un objet de cette classe.** Il y a un autre piège qu'il faut éviter. Supposez que vous écriviez le code suivant :

```
pC1V1= & C5V2 ;
  pC1V1->g() ;
```

Après tout, le pointeur `pC1V1` pointe sur un objet de la classe 5 et celle-ci possède une fonction polymorphique `g`. Or, ce code ne compile pas. Pourquoi ? Tout simplement parce que **le polymorphisme s'applique sur une même chaîne d'héritages**, autrement dit le pointeur `pC1V1` n'étant pas un pointeur sur une des classes contenant une version de la méthode virtuelle `g` l'appel ne peut pas marcher. **Ici, ce sont les méthodes qui sont polymorphiques et non pas les objets !**

10.2.3. Influence des contrôles d'accès

Que se passe-t-il si le contrôle d'accès défini sur une version d'une méthode virtuelle est différent de celui-ci défini dans la classe mère ? Considérez l'exemple suivant.

```
1  #include<stdio.h>
2
3  class Cmere
4  {
5   public :
6            virtual void Afficher() {printf("Objet de la classe Cmere\n");}
7  };
8
9  class Cfille : public Cmere
10 {
11  private :
12           virtual void Afficher() {printf("Objet de la classe Cfille\n");}
13 };
14
15
16 void main()
17 {
18  Cmere * pMERV1 ;
19
20  Cfille FILV3 ;
```

```
21
22  pMERV1 = & FILV3; // pMERV1 pointe sur l'objet FILV3
23  pMERV1->Afficher();
24 }
```

De façon surprenante, la ligne 23 compile et provoque l'exécution de la ligne 12. Par contre, l'appel

```
FILV3.Afficher();
```

ne compile pas. Pourquoi ? Et bien tout simplement parce que dans ce dernier cas c'est un linkage statique qui est réalisé et ce sont donc les restrictions d'accès classiques qui sont appliquées. Dans le cas de la ligne 23, **c'est un linkage dynamique qui est réalisé et le compilateur ne sachant pas à l'avance quelle version de la méthode virtuelle va être appelée il suppose que son contrôle d'accès est le même que celui de la classe associée au pointeur** (c'est-à-dire `Cmere` dans notre exemple).

10.2.4. Le cas des valeurs par défaut

La gestion des valeurs par défaut pour les arguments d'une méthode virtuelle est plus "chaotique". En effet, c'est à la compilation que sont utilisées ou non les valeurs par défaut que vous avez indiquées pour les arguments de votre méthode, selon l'appel que vous en faites. Considérez l'exemple de la section 10.2.1. dans lequel nous ajoutons des valeurs par défaut pour la méthode virtuelle.

```
1  #include<stdio.h>
2
3  class Cmere
4  {
5   public:
6           virtual void Afficher(int iP1=0)
7                   {printf("Objet de la classe Cmere(%d)\n", iP1);}
8  };
9
10 class Cfille : public Cmere
11 {
12  public:
13          virtual void Afficher(int iP1=10)
14                  {printf("Objet de la classe Cfille(%d)\n", iP1);}
15 };
16
17
18
19 void main()
20 {
```

```
21  Cmere * pMERV1;
22
23  Cmere MERV2;
24  Cfille FILV3;
25
26  pMERV1 = & MERV2; // pMERV1 pointe maintenant sur l'objet MERV2
27  pMERV1->Afficher();
28  pMERV1 = & FILV3; // pMERV1 pointe maintenant sur l'objet FILV3
29  pMERV1->Afficher();
30 }
```

L'affichage résultant est le suivant...

```
Objet de la classe Cmere(0)
Objet de la classe Cfille(0)
```

... ce qui montre bien que l'argument par défaut reste 0 même dans le cas de la ligne 29. On voit bien ici le décalage qu'il y a entre les choix faits par le compilateur lors du linkage statique et le linkage dynamique. Lors de la compilation, le compilateur détermine sur la ligne 29 qu'il faut mettre la valeur par défaut pour la méthode `Afficher`, or il ne sait pas quelle version de la méthode sera appelée puisque c'est le linkage dynamique (lors de l'exécution du programme) qui le déterminera. **Le compilateur décide donc dans ce cas que c'est la valeur par défaut de la classe associée au pointeur qui sera systématiquement utilisée**. Ce raisonnement implique également qu'on a le même résultat si aucune valeur par défaut n'est précisée sur la ligne 13.

Voici maintenant une question piège : a-t-on le même comportement si aucun argument n'est passé à la version de la ligne 13 ? Et bien non ! Dans ce cas la ligne 29 provoque l'appel de la méthode de la ligne 6... ce qui est logique puisqu'en ne mettant pas la même interface pour la méthode virtuelle vous avez bloqué le mécanisme du polymorphisme (cf. section 10.2.1. pour un rappel sur les règles à respecter pour mettre en œuvre le polymorphisme).

10.2.5. Les méthodes virtuelles pures et les classes abstraites

En langage C++, une méthode virtuelle *pure* est une méthode virtuelle ne devant pas être définie dans la classe de base (la première classe de la chaîne d'héritages qui la déclare). Cette méthode est donc juste déclarée mais pas définie dans cette classe. Cela peut parfois être utile si vous ne pouvez attacher aucun sens à cette méthode virtuelle dans la classe de base, la méthode prenant un sens dans les classes dérivées. Pour déclarer qu'une méthode est une méthode virtuelle pure, il suffit de faire suivre sa déclaration de "=0".

```
1  #include<stdio.h>
2
```

```
3  class Cmere
4  {
5   public :
6              virtual void Afficher() =0;
7  };
8
9  class Cfille : public Cmere
10 {
11  public :
12             virtual void Afficher()
13                    {printf("Objet de la classe Cfille\n");}
14 };
15
16
17
18 void main()
19 {
20  Cmere * pMERV1;
21
22  Cfille FILV3;
23
24  pMERV1 = & FILV3; // pMERV1 pointe maintenant sur l'objet FILV3
25  pMERV1->Afficher();
26 }
```

Dans cet exemple, la méthode `Afficher` de la classe `Cmere` est une méthode virtuelle pure. Par ailleurs, **elle doit nécessairement être déclarée et définie dans la classe `Cfille` pour pouvoir compiler les lignes 24 et 25.**

Dans cet exemple, si vous essayez de déclarer dans la fonction `main` un objet de la classe `Cmere` vous allez obtenir un message d'erreur vous indiquant que cette classe est abstraite. En effet, *une classe contenant au moins une méthode virtuelle pure est appelée classe abstraite et il n'est pas possible de déclarer des objets de cette classe*. Cela est dû au fait qu'une méthode virtuelle pure n'est pas définie mais juste déclarée et qu'il n'est pas possible de l'appeler via un objet de la classe. Le compilateur interdit donc la création de ces objets.
Notez bien que si vous ne définissez pas le code d'une méthode virtuelle pure dans une classe dérivée alors cette classe dérivée est, de fait, une classe abstraite !

Chapitre 11

Les flots

11.1 GÉRER VOS ENTRÉES/SORTIES AVEC LES FLOTS

On appelle *opération d'entrée/sortie* une opération (instruction en langage C++) qui provoque la lecture/écriture de données dans l'environnement du programme. Par exemple, lorsque vous écrivez un message à l'écran vous réalisez une opération d'écriture sur l'écran. En langage C, pour réaliser cette opération de sortie sur l'écran vous utilisez une commande, en l'occurence la commande `printf`.

Le langage C++ introduit un mécanisme différent pour réaliser toutes sortes d'entrées/sorties : le mécanisme des flots. Un flot désigne un canal sur lequel on peut écrire ou lire des données, ce canal étant relié pour une part à votre programme et pour l'autre à l'écran, un fichier... Un flot est représenté par un objet dans lequel on écrit pour écrire des données dans le canal et dans lequel on lit pour lire les données contenues dans le canal. La figure 11.1 synthétise tout cela.

Notez bien que plusieurs flots peuvent être reliés au même périphérique : notamment, plusieurs flots sont reliés à l'écran et permettent d'écrire dessus de différentes façon.
Tous les flots n'ont pas les mêmes caractéristiques :

– *Il y a les flots d'entrée, les flots de sortie et les flots d'entrée/sortie.* Les premiers permettent de lire des données à partir d'un périphérique (par exemple, lire ce qui est tapé sur un clavier), les seconds permettent d'écrire sur un périphérique (par exemple, écrire une phrase à l'écran) et les troisièmes permettent de faire les deux.

– *On peut écrire/lire des données dans un flot en mode texte ou en mode binaire.* Un flot qui fonctionne en mode texte va interpréter la séquence d'octets qui lui est passée comme si c'était une chaîne de caractères, c'est-à-dire que certains caractères comme par exemple"\n" vont être exécutés pour ce qu'ils sont (ici un retour chariot).
On parle parfois d'un *mode de transfert formaté*. En mode binaire, les octets sont transférés sans être interprétés : on parle parfois d'un mode de transfert *brut* ou d'un *mode de transfert non formaté*.

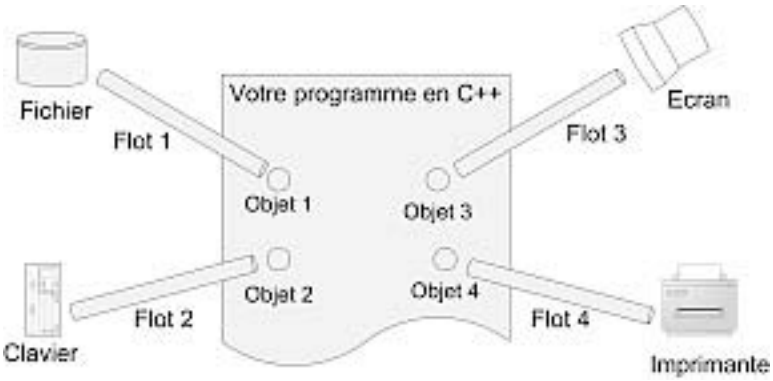


FIG. 11.1: Les flots, les objets et vos programmes

L'avantage des flots est lié aux recommandations du génie logiciel. À savoir, qu'ils proposent une encapsulation des entrées/sorties afin de rendre le programme indépendant des contraintes liées aux périphériques.

Ainsi, pour illustrer simplement, si demain votre écran est changé vous n'avez pas à changer les lignes de code qui permettent de réaliser des affichages puisque celles-ci écrivent dans un objet. Seule la définition de la classe de cet objet est éventuellement à changer. Cela rend votre programme plus modulaire et plus propre et homogénéise également les opérations d'entrées/sorties : que ce soit pour écrire dans un fichier ou écrire à l'écran les opérations ont toutes la même syntaxe et le même principe de fonctionnement.

L'objectif de ce chapitre est uniquement de vous familiariser avec les flots et de vous apprendre à vous en servir.

11.2 CLASSES D'ENTRÉES/SORTIES

Avant de rentrer dans le détail des classes d'entrées/sorties mises à disposition par le langage C++, voyons quatre flots très classiques.

– *cin*. Il s'agit d'un flot d'entrée relié au clavier et qui permet de récupérer ce que l'utilisateur saisi. Il s'agit d'un flot texte. En quelque sorte, récupérer des octets par ce flot est équivalent à la commande `scanf` du langage C.

– *cout*. Il s'agit d'un flot de sortie relié à l'écran et qui permet d'afficher du texte. Il s'agit d'un flot texte. Écrire sur ce flot est équivalent à la commande `printf` du langage C.

– *cerr*. Il s'agit d'un flot de sortie relié à l'écran et qui permet d'afficher des messages d'erreur. Il s'agit d'un flot texte non bufferisé, c'est-à-dire que tout octet écrit dans le flot est immédiatement écrit à l'écran.

– *clog*. Il s'agit d'un flot de sortie relié à l'écran et qui permet d'afficher des messages d'erreur. Il s'agit d'un flot texte bufferisé, c'est-à-dire que tout octet écrit dans le flot est écrit par paquets à l'écran.

Pour manipuler ces flots vous pouvez utiliser les opérateurs d'insertion («) et d'extraction (») qui ont été surchargés pour chacune des classes d'appartenance de ces objets. Voici un exemple ci-dessous (remarquez que ces flots sont définis dans l'interface `iostream.h`).

```
1  #include<iostream.h>
2
3  void main()
4  {
5   int iAge;
6
7   cout << "Bonjour cher lecteur\n" << "Merci de saisir votre âge : ";
8   cin >> iAge;
9   cout << "Votre âge est de " << iAge << " ans\n";
10 }
```

L'écriture d'un texte à l'écran se fait donc en envoyant des données dans le flot `cout` à l'aide de l'opérateur d'insertion «. Cet opérateur retournant une référence sur le flot sur lequel il est appelé vous pouvez donc enchaîner les appels à cet opérateur sur une même ligne (cf. ligne 7 et 9).
La lecture d'un texte tapé au clavier est réalisée en consultant le flot `cin` à l'aide de l'opérateur d'extraction ». Comme pour l'opérateur d'insertion, l'opérateur d'extraction retourne une référence sur le flot et peut donc être utilisé en cascade sur une même ligne.

Notez bien que dans l'exemple suivant *aucun indicateur de format n'est indiqué aux opérateurs, contrairement à ce que nous aurions eu à faire avec les commandes* `printf` *et* `scanf`. Prenons en exemple la ligne 8. À aucun moment nous n'indiquons que la variable `iAge` est de type `int` (pas d'indicateur de format du type "`%d`") : c'est la surcharge de l'opérateur » qui va récupérer la chaîne de caractères tapée par l'utilisateur puis déterminer le type de la variable dans lequel le résultat doit être mis. Ici le type est `int`, donc l'opérateur réalise une conversion du type `char *` vers le type `int` et met le résultat dans `iAge`. *Le principe reste exactement le même pour l'objet* `cout` *et l'affichage de la variable* `iAge`.

Ces quatre objets vous montrent le principe de fonctionnement des flots. Rentrons maintenant un peu plus dans l'organisation générale des flots. Plusieurs classes sont disponibles dans la librairie standard du langage C++, toutes sont dérivées du patron de classe `ios_base` (figure 11.2).

La figure 11.2 vous montre bien comment l'architecture des flots peut être complexe : et encore, cette figure ne vous en montre qu'une petite partie ! Globalement, les flots sont organisés autour de la classe de base `ios_base` qui contient les attributs et les méthodes indépendants des patrons de classes définis ensuite. Différents patrons sont ensuite dérivés de cette classe, à différents niveaux. Notamment le patron `basic_iostream` est dérivé des patrons `basic_istream` et `basic_ostream`. Il permet de créer des flux d'entrées/sorties tandis que les deux derniers permettent de créer des flux d'entrée et de sortie, respectivement. **L'objet `cin` est un objet de la classe `istream` créée par instanciation du patron `basic_istream`**. De même, **les objets `cout`, `cerr` et `clog` sont des objets de la classe `ostream` créée par instanciation du patron `basic_ostream`**.
La gestion des fichiers est rendue possible par le patron `basic_fstream` et la classe `fstream` créée par instanciation. En créant des objets de cette classe vous pourrez manipuler des fichiers sur disque comme vous le faisiez en langage C avec les commandes `fopen`, `fprintf`... Je reviendrais sur la gestion des fichiers un peu plus loin dans ce chapitre dans la section 11.2.3.

11.2.1. Écrire à l'écran : le flot `cout`

Nous avons déjà abordé dans la section précédente comment écrire à l'écran avec le flot `cout` : il suffit d'utiliser l'opérateur d'insertion "«". Voyons maintenant un peu plus en détail comment cela fonctionne.

L'objet `cout` est un objet de la classe `ostream` (défini dans le fichier `iostream.h`) au sein duquel l'opérateur `operator«` a été surchargé pour tous les types de base : `short`, `int`, `float`... Ainsi la ligne :

```
cout « 5 ;
```

est équivalente à :

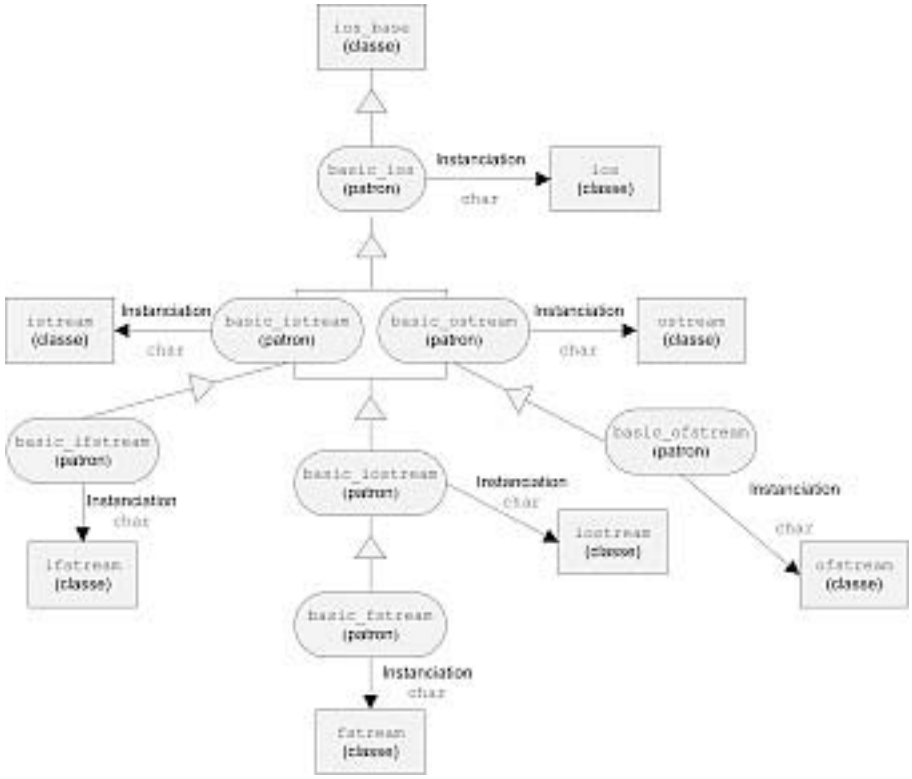


FIG. 11.2: Organisation des classes permettant l'accès aux flots

```
cout.operator«(5);
```

et utilise la surcharge de l'opérateur "«" qui prend en paramètre un `int`. Si vous avez compris le principe de fonctionnement il n'est nul besoin d'aller plus loin pour comprendre comment utiliser le flot de sortie pour écrire à l'écran. Si vous souhaitez rentrer encore plus dans les détails vous pouvez vous référer à la référence [**3**].

Pour compléter cette présentation générale sur l'utilisation du flot `cout`, voici une liste de méthodes de la classe `ostream` qui sont appelables à partir de `cout` (en complément de la surcharge de `operator«`) :

– `put(char c)`. Cette méthode écrit un caractère dans le flot qui est renvoyé par référence.

– `write(char *pc, int i)`. Cette méthode écrit une chaîne de caractères dans le flot qui est renvoyé par référence. Notez bien que cette méthode permet d'écrire des octets sur le flot sans gérer leur formatage (pas de gestion des retours chariots...) ce qui permet d'écrire en mode "brut" dans le flot.

Tout ce que nous avons vu ici est également applicable aux flots `cerr` et `clog`. Notez bien que ce dernier est bufferisé, c'est-à-dire que si, par exemple vous écrivez `clog.put("t")` il est possible que le caractère ne soit pas affiché immédiatement à l'écran. L'exemple suivant vous montre comment faire pour provoquer l'affichage immédiat : `clog.put("t")«flush;`
Le mot-clef `flush` correspond à un manipulateur sur le flot. Il en existe d'autres comme par exemple `endl`, qui provoque un retour à la ligne (équivalent de '$\backslash n$' dans la commande `printf`).

11.2.2. Lire à l'écran : le flot `cin`

Les principes d'utilisation du flot `cin` sont similaires à ceux exposés dans la section précédente sur le flot `cout`. L'opérateur qui permet de récupérer les informations saisies au clavier par l'utilisateur est l'opérateur d'extraction "»". L'objet `cin` est un objet de la classe `istream` (définit dans le fichier `iostream.h`) au sein duquel l'opérateur `operator»` a été surchargé pour tous les types de base : `short`, `int`, `float`... Le principe est ici strictement le même que pour les flots de la classe `ostream`. Quand vous écrivez la ligne :

```
cin » i;
```

avec `i` une variable de type `int`, le compilateur appelle la surcharge de l'opérateur `operator»` qui prend en paramètre un `int` et convertit la chaîne de caractères saisie par l'utilisateur à l'écran en un objet de ce type.

```
1 #include<iostream.h>
2
3 void main()
4 {
5  int iV1,iV2,iV3;
6
7  cin >> iV1 >> iV2 >> iV3;
8  cout << "Les valeurs saisies sont : " << iV1 << iV2 << iV3 << endl;
9 }
```

Sur l'exemple précédent, la ligne 7 provoque l'enchaînement de trois appels à la surcharges de l'opérateur `operator»`, ce qui implique que si vous tapez sur le clavier "3 4 5" puis sur la touche "Entrée" vous aurez alors `iV1=3`, `iV2=4` et `iV1=5`.

Lors de l'acquisition avec l'opérateur '»', la fin de saisie est obtenue en appuyant sur la touche "Entrée". Dans le cas d'une acquisition multiple (plusieurs variables), l'espace est considéré comme une délimiteur (cf. exemple précédent).

Il existe également une liste de méthodes de la classe `istream` qui sont appelables à partir de l'objet `cin` (en complément de la surcharge de `operator»`) :

– `get(char c)`. Cette méthode lit un caractère dans le flot qui est renvoyé par référence.
– `getline(char *pc, int i, char c)`. Cette méthode lit une chaîne de `(i-1)` caractères dans le flot à moins que le caractère *délimiteur* `c` ne soit rencontré avant le `(i-1)`ème caractère. Les caractères lus sont stockés à l'adresse `pc` et le caractère '\ 0' est ajouté en fin de chaîne. Cette méthode renvoie le flot par référence.
– `gcount()`. Cette méthode retourne le nombre de caractères effectivement lus lors du dernier appel à une méthode de lecture comme `getline`.
– `read(char *pc, int i)`. Comme la méthode `getline`, cette méthode permet de lire des caractères saisis au clavier par l'utilisateur et les stocke à l'adresse `pc`. Notez bien que cette méthode ne fait aucune supposition sur la nature des caractères lus et qu'aucun délimiteur n'est pris en compte comme cela était le cas dans la méthode `getline`.

11.2.3. Manipuler des fichiers

En plus des classes d'entrée et de sortie précédentes, on dispose de classes spécialement conçues pour réaliser l'échange de données entre les fichiers et le programme :

– `ifstream` pour la gestion de fichiers en lecture,
– `ofstream` pour la gestion de fichiers en écriture,
– `fstream` pour la gestion de fichiers en écriture et en lecture.

Ces classes, qui sont définies dans le fichier `fstream.h`, possèdent un certain nombre d'attributs relatifs à la gestion des fichiers : une *longueur* qui correspond au nombre de caractères présents dans le flot (et cela peut être différent du nombre de caractères contenus dans le fichier), un *début* qui correspond au premier caractère situé dans le flot, une *fin* qui correspond à la position après le dernier caractère dans le flot et enfin une *position courante* qui correspond à la position du caractère qui va être écrit/lu par les opérations sur le flot.

Nous allons maintenant voir plus en détail comment ouvrir et manipuler des fichiers à l'aide des flots.

a) Ouvrir un fichier en lecture seule : la classe `ifstream`

Pour ouvrir un fichier en lecture seule il vous suffit simplement de créer un objet de la classe `ifstream` en indiquant, comme paramètre lors de l'initialisation lors de la déclaration, le nom du fichier. Sans autre indication de votre part, le fichier est ouvert en mode texte.

```
1 /* Dans cet exemple, je suppose l'existence d'un fichier c:\phrase.txt
2    dans lequel figure la ligne :
```

```
3     J'aime le C++ */
4
5  #include<fstream.h>
6
7  void main()
8  {
9   ifstream MonFichier("c :\\phrase.txt");
10  char cLigne[255];
11
12  MonFichier >> cLigne;
13
14  cout << "La phrase contenue dans le fichier est : " << cLigne << endl;
15 }
```

Dans cet exemple, notez bien la syntaxe de la ligne 9. Un objet de la classe `ifstream`, nommé `MonFichier`, est déclaré et initialisé par appel à un constructeur à un argument prenant en paramètre une chaîne de caractère : le nom du fichier à ouvrir. Comme pour la commande `fopen` du langage C, vous remarquez que si vous devez indiquer le chemin d'accès au fichier (comme c'est le cas ici) alors les caractères "\" sont doublés.

L'affichage provoqué à l'écran est, presque de façon surprenante, le suivant : *J'aime*. En effet, toute la ligne n'est pas lue dans le fichier puisque comme pour la classe `istream` le caractère espace est un séparateur. Ainsi dans la surcharge de l'opérateur "»" pour la classe `ifstream`, la lecture s'arrête sur ce séparateur.

Que se passe-t-il si jamais le fichier n'existait pas ? Et bien tout simplement rien sur la ligne 9 : le fichier semble avoir été ouvert mais pourtant ce n'est pas le cas et aucune remontée d'exception n'a lieu. Vous devez donc tester, après ouverture que le fichier est correctement ouvert. Vous avez trois possibilités :

(1) Utilisez la méthode `is_open()` de la classe `ifstream`. Il vous suffit de tester si l'appel `MonFichier.is_open()` sur la ligne 11 retourne la valeur `true`. Si c'est le cas le fichier est bien ouvert.

(2) Utilisez la méthode `fail()` de la classe `ifstream`. Il vous suffit de tester si l'appel `MonFichier.fail()` sur la ligne 11 retourne la valeur `false`. Si c'est le cas le fichier est bien ouvert.

(3) Utilisez la surcharge de l'opérateur de négation "!". Il vous suffit de tester si l'appel `!MonFichier` sur la ligne 11 retourne la valeur `false`. Si c'est le cas le fichier est bien ouvert.

Notez bien que le fichier peut être fermé en appelant la méthode `close` sur le flot. Il peut être également ouvert en appelant la méthode `open` (cette méthode est équivalente à l'appel du constructeur de la classe `ifstream`).

b) Ouvrir un fichier en écriture : la classe `ofstream`

Pour ouvrir un fichier en écriture il vous suffit simplement de créer un objet de la classe `ofstream` en indiquant, comme paramètre lors de l'initialisation lors de la déclaration, le nom du fichier. Sans autre indication de votre part, le fichier est ouvert en mode texte.

```
1 #include<fstream.h>
2
3 void main()
4 {
5  ofstream MonFichier("c :\\phrase.txt");
6
7  MonFichier << "J'aime le C++" << endl;
8
9 }
```

La syntaxe de la ligne 5 est similaire au cas de l'ouverture d'un fichier en lecture. *Que se passe-t-il si jamais le fichier n'existait pas ?* Et bien il serait créé vide.

Notez bien que le fichier peut être fermé en appelant la méthode `close` sur le flot. Il peut être également ouvert en appelant la méthode `open` (cette méthode est équivalente à l'appel du constructeur de la classe `ifstream`).

c) Paramétrer l'ouverture et la gestion des fichiers

Il est possible en langage C++ de spécifier la façon dont le fichier est ouvert. Par défaut, il est ouvert en mode texte que ce soit en écriture ou en lecture. Pour changer le mode d'ouverture d'un fichier vous devez passer un second argument lors de l'initialisation de l'objet (le premier étant le nom du fichier). Ce second argument correspond à des valeurs de bits sur lesquels vous devez faire des masques logiques. La liste des masques que vous pouvez appliquer, ceux-ci étant hérités de la classe `ios` vers les classes `ifstream`, `ofstream` et `fstream` (ce sont des attributs de la classe `ios`), est présentée dans le tableau 11.1.

Ces masques s'utilisent en réalisant un OU logique pour changer la valeur du mode d'ouverture par défaut. Voici un exemple qui vous illustrera clairement comment faire.

```
1 #include<fstream.h>
2
3 void main()
4 {
5  ofstream MonFichier("c :\\phrase.txt", ios : :binary | ios : :ate);
6  ...
7 }
```

Nom du masque	Signification
`ios::app`	Avant chaque opération d'**écriture** le flot est positionné sur la fin du fichier (mode append).
`ios::ate`	Après l'ouverture du fichier en **écriture** le flot est positionné sur la fin du fichier (mode at end).
`ios::binary`	Le fichier est ouvert en mode binaire et non plus en mode texte.
`ios::in`	Le fichier est ouvert en lecture seule (utilisable avec la classe `fstream`).
`ios::out`	Le fichier est ouvert en écriture seule (utilisable avec la classe `fstream`).
`ios::trunc`	Le fichier, s'il existe, est vidé de son contenu.
`ios::nocreate`	Le fichier n'est ouvert que s'il existe, sinon l'opération échoue.
`ios::noreplace`	Le fichier n'est ouvert que s'il n'existe pas, sinon l'opération échoue.
`ios::beg`	Le déplacement dans le fichier en accès direct se fait à partir du début du fichier.
`ios::cur`	Le déplacement dans le fichier en accès direct se fait à partir de la position courante dans le fichier.
`ios::end`	Le déplacement dans le fichier en accès direct se fait à partir de la fin du fichier.

TAB. 11.1: Liste des masques utilisables dans la gestion des fichiers

Dans cet exemple, le fichier `phrase.txt` est ouvert en mode binaire avec ajout en fin de fichier.

Il est également possible d'accéder directement à un fichier en lecture ou en écriture. La classe `ios` fournit des masques permettant d'exprimer les déplacements à l'intérieur des fichiers. Les valeurs sont indiquées dans le tableau 11.1 : `beg`, `cur` et `end`. Ces masques sont utilisés dans les méthodes `seekg` (classe `ifstream`) et `seekp` (classe `ofstream`) pour préciser un déplacement dans le fichier ouvert. Il est également possible de connaître la position courante dans le fichier à l'aide des méthodes `tellg` (classe `ifstream`) et `tellp` (classe `ofstream`). En voici un exemple. L'affichage provoqué par la ligne 17 est "C". Notez bien que la valeur du déplacement indiqué aux méthodes `seekp` et `seekg` peut être négatif : dans ce cas cela signifie que vous allez effectuer un déplacement en remontant dans le fichier. Cette valeur représente le nombre d'octets duquel on se déplace.

```
1  /* Dans cet exemple, je suppose l'existence d'un fichier c :\phrase.txt
2     dans lequel figure la ligne :
3     J'aime le C++ */
4
5  #include<fstream.h>
6
7  void main()
8  {
9   ifstream MonFichier("c :\\phrase.txt");
10  char cLigne[255];
```

```
11
12  cout << "Position courante dans le fichier : "<<MonFichier.tellg() << endl;
13  MonFichier >> cLigne;
14  cout << "Le début de phrase est : " << cLigne << endl;
15  MonFichier.seekg(3,ios : :cur);
16  MonFichier >> cLigne;
17  cout << "La fin de phrase est : " << cLigne << endl;
18 }
```

Vous pouvez accéder à d'autres fonctionnalités sur les flots associés aux fichiers en regardant dans l'aide de votre environnement de développement les méthodes des classes `ifstream`et `ofstream`.

11.3 STATUTS D'ERREUR SUR UN FLOT

À chaque flot est associé un ensemble de bits d'erreur. Ces bits, interprétés comme un entier, forment le statut d'erreur du flot. Quatre valeurs, définies dans la classe `ios` sont intéressantes (tableau 11.2).

Si le flot est dans un état d'erreur, les opérations ne peuvent pas s'effectuer tant que l'erreur n'a pas été corrigée (indépendemment du flot) et que le bit d'erreur correspondant n'a pas été désactivé. Pour réaliser la seconde condition, on dispose de méthodes appartenant à la classe `ios` pour connaître la valeur d'un bit, et pour la modifier. Ces méthodes sont les suivantes :

- `eof` : renvoie l'état du bit `eofbit`,
- `bad` : renvoie l'état du bit `badbit`,
- `fail` : renvoie l'état du bit `failbit`,
- `good` : renvoie 1 si aucun des trois bits précédents n'est activés,
- `rdstate` : renvoie le statut d'erreur du flot,
- `clear` : active le bit passé en paramètre, et met tous les autres à 0. On fixe en fait le statut d'erreur du flot.

Nom du masque	Signification
`ios : :goodbit`	Ce bit vaut 0 s'il n'y a pas d'erreur sur le flot.
`ios : :eofbit`	Ce bit vaut 1 si la fin du flot est atteinte.
`ios : :failbit`	Ce bit vaut 1 si la prochaine opération d'entrée/sortie ne peut pas s'effectuer.
`ios : :badbit`	Ce bit vaut 1 si le flot est dans un état irrécupérable.

TAB. 11.2: Liste de certains statuts d'erreurs sur les flots

Si l'on souhaite activer un bit sans modifier la valeur des autres, on passera en paramètre à la fonction `clear` le nom du bit à activer et le résultat de l'appel de la fonction `rdstate` séparés par le caractère "|". Ainsi on affecte l'ancien statut d'erreur plus un bit particulier.

L'état d'erreur d'un flot peut être réinitialisé à 0 à l'aide de la méthode `clear`.

```
1  /* Dans cet exemple, je suppose que le fichier c :\phrase.txt
2     est un fichier vide */
3
4  #include<fstream.h>
5
6  void main()
7  {
8   ifstream MonFichier("c :\\phrase.txt");
9   char cLigne[255];
10
11   MonFichier >> cLigne;
12
13   // Je teste l'état du flot
14   if (MonFichier.eof())
15   {
16           cout << "Le fichier est vide" << endl;
17           return;
18   }
19
20   cout << "La phrase contenue dans le fichier est : " << cLigne << endl;
21 }
```

Vous remarquerez dans cet exemple que le flag `eofbit` est testé après l'opération de lecture. En effet, c'est celle-ci (et non l'ouverture du fichier) qui provoque la lecture de la fin du fichier (*eof*).

Il est également possible d'utiliser le *mécanisme des exceptions* pour la remontée des erreurs. Par défaut, ce mécanisme n'est pas activé sur les flots mais peut l'être grâce à l'appel de la méthode `exceptions` sur l'objet. Le paramètre accepté par cette méthode doit être un OU logique des masques associés aux bits pour lesquels on souhaite une remontée d'exceptions. L'objet levé est alors du type `ios::failure`. Reprenons l'exemple précédent.

```
1  /* Dans cet exemple, je suppose que le fichier c :\phrase.txt
2     est un fichier vide */
3
4  #include<fstream.h>
5
```

```
6  void main()
7  {
8   ifstream MonFichier("c :\\phrase.txt");
9   char cLigne[255];
10
11  // J'active la remontée d'exceptions dans le cas où eofbit ou badbit sont mis à 1
12  MonFichier.exceptions(ios : :eofbit | ios : :badbit);
13
14  MonFichier >> cLigne;
15
16  cout << "La phrase contenue dans le fichier est : " << cLigne << endl;
17 }
```

La ligne 12 indique au compilateur que les erreurs `eofbit` et `badbit` sur l'objet `MonFichier` doivent être signalées par la levée d'une exception. C'est ce qui sera fait suite à l'exécution de la ligne 14. **Notez bien que cette possibilité, bien que figurant dans la norme ANSI du C++ n'est pas offerte par tous les compilateurs C++.**

> Aussi souvent que possible vous privilégierez l'utilisation des exceptions pour la gestion des erreurs sur les flots (au minimum les erreurs basiques de lecture/écriture, la gestion du cas `eof` n'étant pas une nécessité).

11.4 FORMATER VOS FLOTS

J'entends par *formatage d'un flot* la faculté que vous avez de pouvoir régler le contenu d'une suite de caractères envoyés sur celui-ci. Le formatage est géré par un mot d'état qui est constitué d'un ensemble de bits : en fonction de leur activation ceux-ci commandent telle ou telle opération à effectuer sur le contenu du flot. Vous avez déjà l'habitude de manipuler le formatage, en langage C : lorsque vous utilisez la commande `printf`, par exemple, vous devez indiquer le type des variables que vous voulez faire afficher[1]. En langage C++ deux possibilités s'offrent à vous (contrairement au langage C) : soit vous ignorez totalement le formatage et vous vous contentez de celui par défaut (c'est ce que nous avons fait jusqu'à présent dans les exemples de ce chapitre), soit vous précisez le formatage que vous souhaitez pour les variables à envoyer sur le flot.

Le formatage fait appel à des symboles, appelés *manipulateurs*, qui permettent de modifier le mot d'état d'un flot. Ceux-ci renvoient un flot, dont le fonctionnement est modifié par leur application. Deux types de manipulateurs existent, en fonction

1. %d pour `int`...

de la présence ou non de paramètres. Nous donnons dans un premier temps leurs prototypes.

```
istream & nom_manipulateur()
ostream & nom_manipulateur()
istream & nom_manipulateur(argument)
ostream & nom_manipulateur(argument)
```

La liste de certains manipulateurs est indiquée dans le tableau 11.3.

Nom du manipulateur	Signification
`ios::dec`	Dans la suite du flot la numération est en base 10.
`ios::hex`	Dans la suite du flot la numération est en base 16.
`ios::oct`	Dans la suite du flot la numération est en base 8.
`ios::endl`	Provoque un retour chariot.
`ios::ends`	Provoque une fin de chaîne.
`ios::flush`	Vide le flot.
`ios::ws`	Dans la suite du flot les espaces sont ignorés.
`ios::setbase(int)`	Définit la base conversion dans la suite du flot.
`ios::setprecision(int)`	Précise le nombre de décimaux envoyés dans les flottants passés dans la suite du flot.
`ios::setw(int)`	Précise la largeur de la chaîne envoyée sur le flot (typiquement un nombre de caractères affichés).

TAB. 11.3: Liste des manipulateurs utilisables dans le formatage des flots

Voici un exemple d'utilisation des manipulateurs.

```
1 #include<fstream.h>
2
3 void main()
4 {
5
6  cout<< "Affichage en base 8 : " << oct << 55 << endl;
7  cout<< "Affichage en base 16 : " << hex << 55 << endl;
8  cout<< "Affichage en base 10 : " << dec << 55 << endl;
9 }
```

L'affichage est le suivant.

```
Affichage en base 8 : 67
Affichage en base 16 : 37
Affichage en base 10 : 55
```

EXERCICES CORRIGÉS

EXERCICES CORRIGÉS

Vous trouverez dans cette annexe un ensemble d'exercices à réaliser, sur papier ou sur machine, vous permettant de vérifier si vous avez assimilé les notions vues dans cet ouvrage. Notez bien que ne figurent pas ici des exercices pour tous les chapitres : je n'y fais figurer que ceux que je juge fondamentaux pour une pratique basique du langage C++.

Une dernière remarque : dans les portions de code qui sont proposées au travers des différents exercices ne sont jamais mentionnées les inclusions de librairies systèmes nécessaires à la bonne compilation. Je suppose, néanmoins, que ces inclusions nécessaires sont réalisées.

CHAPITRE 2

Exercice 1. Déclaration et initialisation des variables

Le code suivant est-il correct, autrement dit est-il compilable ?

```
1  void main()
2  {
3   int iBoucle;
4
5   for (iBoucle=0;iBoucle<10;iBoucle++)
6   {
7    int iBoucle2;
8    iBoucle2=iBoucle+1;
9   }
10  printf("Valeur de iBoucle2 : %d\n",iBoucle2);
11 }
```

Si cela n'est pas le cas vous indiquerez la (les) ligne(s) qui ne compile(nt) pas. Quelles sont alors les modifications à apporter pour que cela compile ?

Exercice 2. Portée et visibilité des variables

Le code suivant est correct, c'est-à-dire qu'il compile, mais quel est son comportement et qu'affiche-t-il à l'écran ?

```
1  void main()
2  {
3   int iBoucle;
4
5   for (iBoucle=0;iBoucle<3;iBoucle++)
6   {
7    for (int iBoucle=5;iBoucle<7;iBoucle++)
```

```
8           printf("iBoucle (1) = %d\n",iBoucle);
9     printf("iBoucle (2) = %d\n",iBoucle);
10   }
11  }
```

Exercice 3. Portée et visibilité des variables statiques

Le code suivant est correct, c'est-à-dire qu'il compile, mais quel est son comportement et qu'affiche-t-il à l'écran ?

```
1   void main()
2   {
3    int iBoucle;
4
5    for (iBoucle=0;iBoucle<3;iBoucle++)
6    {
7     for (static int iBoucle=5;iBoucle<7;iBoucle++)
8           printf("iBoucle (1) = %d\n",iBoucle);
9     printf("iBoucle (2) = %d\n",iBoucle);
10   }
11  }
```

Exercice 4. Passage d'arguments par référence

Écrivez une fonction, que vous nommerez `f`, qui prend en paramètre un argument par référence de type `int` et qui lui affecte la valeur 10.

Exercice 5. Retour d'arguments par référence

Écrivez une fonction, que vous nommerez `f`, qui ne prend pas de paramètre mais retourne par référence le contenu d'une variable locale à laquelle vous aurez affecté la valeur 10. Écrivez ensuite une fonction `main` qui va affecter la valeur de retour de `f` dans une variable locale.

Quelle remarque pouvez-vous faire sur ce code ? Est-il judicieux ?

Exercice 6. Passage d'arguments par référence : appel de fonction

Voici une fonction nommée `f` qui admet trois paramètres passés par référence, valeur et adresse. La fonction `main` fait appel de plusieurs façons à cette fonction : pour chacun des appels des lignes 12 à 15 précisez s'ils compilent. Si ce n'est pas le cas indiquez-en la cause et les modifications à réaliser pour qu'ils compilent.

```
1  void f(float & fP1, int iP2, float *fP3)
2  {
3   ...
4  }
5
6  void main()
7  {
8   float fV1;
9   int iV2;
10  float fV3;
11
12  f(fV1,iV2,&fV3);
13  f(&fV1,iV2,&fV3);
14  f(fV1,5,fV3);
15  f(4,5,&fV3);
16 }
```

Exercice 7. Variables références

Définissez une fonction, nommée `f`, qui contient la déclaration d'une variable référence pointant sur une autre variable, nommée `iV1`, de type `int`.

CHAPITRES 3 ET 4

Exercice 8. Déclarer, définir et utiliser des classes : la classe `Csac`

On souhaite écrire une classe, nommée `Csac`, qui permet de représenter des sacs dans votre logiciel. Un sac est défini par les attributs suivants :

- un type de sac (défini par un entier dont la valeur fera référence au type de sac : sac plastique, sac-à-dos...),
- le nombre d'éléments présents dans le sac,
- une liste d'éléments présents dans le sac (définit par un tableau de type entier dont chaque élément indique le numéro de l'élément présent dans le sac, par exemple l'élément 5 représente un paquet de mouchoirs).

Par exemple, un objet de la classe `Csac` contenant la valeur 1 pour le premier attribut, 1 pour le deuxième et 5 pour le troisième représente un *sac plastique* contenant uniquement un paquet de mouchoirs.

On souhaite implémenter les méthodes suivantes au sein de la classe :

– `ajouter_element`, permet d'ajouter un élement dans le sac. Cette méthode prend un paramètre du type `int` qui est le numéro de l'élément à mettre dans le sac.
– `enlever_element`, permet de retirer le dernier élément ajouté dans le sac. Cette méthode ne prend pas de paramètre mais renvoie le numéro du dernier élément de la liste. Cet élément est retiré de la liste.
– `taille`, permet de connaître le nombre d'éléments présents dans le sac. Cette méthode retourne ce nombre.
– `fixer_type`, permet de fixer le type de sac. Cette méthode prend en paramètre le nouveau type de sac.
– `lire_type`, permet de connaître le type de sac. Cette méthode retourne le type.

Vous avez également quelques impératifs à respecter lors de la création de la classe `Csac` :

(1) Respectez les conventions de nommage pour les méthodes et attributs que vous avez vues dans cet ouvrage.
(2) La liste doit être implémentée sous forme d'un tableau dont la taille est fixée lors de la déclaration.
(3) Tout passage de paramètre à une méthode se fera par référence.
(4) Vous n'intégrerez pas la gestion des exceptions.

Voici les questions auxquelles vous devez répondre :

(1) Écrivez l'interface de la classe `Csac`.
(2) Écrivez le corps de la classe `Csac`.
(3) Écrivez une fonction `main` qui déclare deux objets `sac1` et `sac2` de la classe `Csac`. Ces deux objets doivent être de type différent. Écrivez le code qui remplit le premier sac avec 10 objets numérotés de 1 à 10 puis qui le vide pour remplir le second sac avec ces éléments.
(4) Comment faire pour compter le nombre de sacs créés dans le programme ?

Exercice 9. Déclarer, définir et utiliser des classes : la classe `CListe`

Supposons que dans l'exercice précédent nous ayons voulu utiliser une classe `CListe` pour implémenter la liste des éléments d'un sac. Une liste est définie par les attributs suivants :

– Le nombre d'éléments présents dans la liste.
– La liste des éléments de la liste.

Nous souhaitons que la liste soit définie de façon un peu générique, c'est-à-dire que la liste soit un tableau de type `Telement` où `Telement` fait référence à un type défini par l'utilisateur de votre classe[2]. Le tableau doit être de taille dynamique, c'est-à-dire représenté par un pointeur dont vous allez gérer la réallocation en fonction des ajouts/suppressions d'éléments.

> Je vous rappelle que la fonction `realloc` est définie dans l'unité `stdlib.h` et que sa syntaxe est la suivante :
>
> ```
> void * realloc(void *memblock, size_t size)
> ```
>
> où `memblock` est l'adresse du pointeur déjà alloué (si `memblock=NULL`, la fonction `realloc` se comporte comme `malloc`) et `size` est la nouvelle taille en octets de l'objet pointé. `realloc` renvoie `NULL` si `size=0` ou s'il n'y a pas assez de mémoire disponible pour le nouveau bloc.
> Si `memblock` ne correspond à aucune zone mémoire allouée et est différent de NULL, la fonction `realloc` va lever une exception (*assertion failed*).

(1) Écrivez l'interface de la classe `CListe` (fichier `CListe.h`). Inspirez-vous des méthodes listées dans l'exercice précédent pour en déduire la liste des méthodes à faire figurer dans votre classe `CListe`.

(2) Écrivez le corps de la classe `CListe` (fichier `CListe.cpp`).

Exercice 10. Allocation d'objets dynamiques

Considérez les quatre fonctions suivantes, nommées `f1`, `f2`, `f3` et `f4`.

```
1  void f1()
2  { // Première version
3    Cliste *LISV1;
4
5    LISV1=(Cliste *)malloc(sizeof(Cliste));
6    ...
7  }
8
9  void f2()
10 { // Seconde version
11   Cliste *LISV1;
12
13   LISV1=new Cliste;
14   ...
15 }
```

2. Via la ligne, par exemple dans le cas d'un élément de type entier : `typedef int Telement`.

```
16
17 void f3()
18 { // Troisième version
19   Cliste *LISV1;
20
21   LISV1=(Cliste *)malloc(10*sizeof(Cliste));
22   ...
23 }
24
25 void f4()
26 { // Quatrième version
27   Cliste *LISV1;
28
29   LISV1=new Cliste[10];
30   ...
31 }
```

Expliquez ce que fait chacune de ces fonctions en détaillant le mécanisme d'appel aux constructeurs.

Exercice 11. Allocation d'objets dynamiques : les doubles pointeurs

(1) Expliquez si la fonction `f5`, dont le code est donné ci-dessous, est correct ou pas (est-ce qu'il compile ?). Que fait-elle ?

```
1  void f5()
2  {
3    int iBoucle;
4    Cliste **LISV1;
5
6    LISV1=(Cliste **)malloc(10*sizeof(Cliste *));
7    for (iBoucle=0;iBoucle<10;iBoucle++)
8              LISV1[iBoucle]=new Cliste [20];
9    ...
10 }
```

(2) La version ci-dessous est-elle meilleure ?

```
1  void f5()
2  {
3    int iBoucle;
4    Cliste **LISV1;
5
6    LISV1=new (Cliste *)[10];
```

```
7   for (iBoucle=0;iBoucle<10;iBoucle++)
8           LISV1[iBoucle]=new Cliste [20];
9   ...
10  }
```

Exercice 12. Déclarer, définir et utiliser des classes : la classe `Ccellule`

On s'intéresse maintenant à l'écriture d'une classe permettant de gérer des listes chaînées bidirectionnelles d'éléments. Une telle liste est constituée de cellules liées entre elles par le chaînage : chaque cellule possède un pointeur sur son prédécesseur et un pointeur sur son successeur (figure 11.3). La cellule contient alors les éléments (notés X sur la figure) que vous voulez lui voir stocker. La liste chaînée possède une cellule *tête* (la première cellule) et une cellule *queue* (la dernière cellule). Pour référencer la liste chaînée il suffit de connaître l'adresse de la cellule tête ou de la cellule queue (généralement la cellule tête).

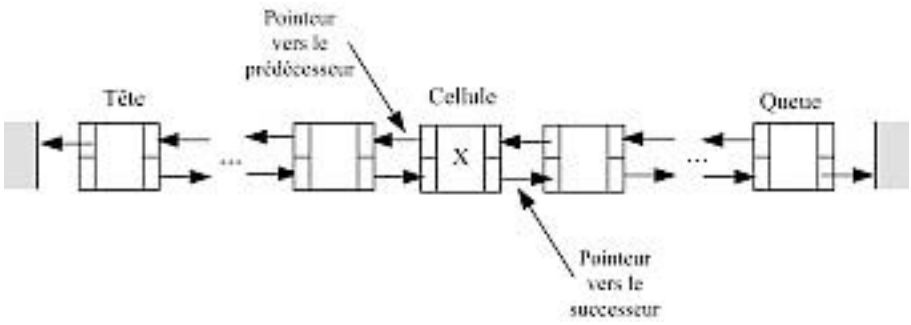


FIG. 11.3: Structure d'une liste chaînée

(1) Écrire l'interface d'une classe `Ccellule` qui représente une cellule d'une liste chaînée bidirectionnelle. Considérez que l'élément contenu dans cette cellule est un entier.

(2) Voici un morceau de code contenu dans la fonction `main`.

```
1  void main()
2  {
3   Ccellule cellule1;
4   ...
5   Ccellule cellule2(cellule1);
6   ...
7   Ccellule cellule3;
```

```
8  cellule3=cellule2 ;
9  ...
10 }
```

On suppose que les lignes "..." correspondent à du code compilable. Le code ci-dessus compile mais il y a potentiellement un plantage aléatoire à l'exécution. D'où peut-il provenir ?

(3) Si vous avez détecté un problème dans la question précédente comment le résoudre ?

(4) Écrire le corps de la méthode de la classe `Ccellule` qui prend en paramètre un objet de type `Ccellule` et qui réalise le chaînage de l'objet en cours à la suite de l'objet passé en paramètre.

CHAPITRES 6 ET 7

Exercice 13. L'héritage simple et les exceptions

Dans cet exercice nous allons nous focaliser sur l'héritage, mais dès que nous aurons des situations d'erreur prévisibles vous aurez à mettre en œuvre des gestions par exception (considérez que vous avez accès à la classe `Cexception` présentée dans l'annexe E). Histoire de vous entraîner...

Continuons sur les listes chaînées telles qu'elles ont été introduites dans l'exercice 12. On suppose maintenant qu'une cellule de la liste ne contient aucun élément : juste des pointeurs vers son prédécesseur et son successeur. La classe `Ccellule` possède donc maintenant :

– 2 *attributs* : `CELPrec` et `CELSuiv` qui sont des pointeurs de type `Ccellule`,

– 4 *méthodes* (hors constructeurs et destructeurs : une méthode pour insérer la cellule dans le chaînage, une méthode pour déchaîner, une méthode pour chercher une cellule qui précède à une position donnée et une méthode pour une chercher une cellule qui suit à une position donnée.

Pour corser un peu le tout, vous devez faire en sorte que les attributs soient accessibles à toute classe fille mais aux autres classes.

(1) Écrire l'interface et le corps de la classe `Ccellule`.

(2) À partir de la classe `Ccellule` comment peut-on créer une liste d'entiers, représentée par une classe `Cliste_entier` ? Écrire l'interface de la classe `Cliste_entier`.

(3) Comment faire en sorte qu'aucune classe dérivée de la classe `Cliste_entier` n'ai accès aux attributs de la classe `Ccellule` ?

Exercice 14. L'héritage simple

On suppose l'existence de trois classes `Cmere`, `Cfille` (qui hérite de `Cmere`) et `Cpetitefille` (qui hérite de `Cfille`). Leur interface est donnée ci-dessous.

```
1  class Cmere
2  {
3   public :
4      int a ;
5      int b ;
6   private :
7      int c ;
8   protected :
9      Cmere() ;
10     ~Cmere() ;
11 } ;
12
13 class Cfille : Cmere
14 {
15  public :
16     int d ;
17  protected :
18     int e ;
19  public :
20     Cfille() ;
21     ~Cfille() ;
22 } ;
23
24 class Cpetitefille : public Cfille
25 {
26  public :
27     int a ;
28     int g ;
29     Cpetitefille() ;
30     ~Cpetitefille() ;
31     void f() ;
32 } ;
```

(1) Les trois classes précédentes sont-elles correctements déclarées, *i.e.* provoquent-elles des erreurs de compilation ?

(2) Faites un tableau récapitulatif qui précise pour chaque classe et pour chaque attribut quel est son statut, son accès interne et son accès externe.

(3) On suppose que dans la méthode `f` on trouve la ligne suivante : `a=d ;`.
Quel attribut `d` est référencé ? Quel attribut `a` est référencé ? Comment référencer l'autre attribut `a` ?

(4) On considère les fonctions `f1` et `f2` suivantes :

```
1  void f1(Cpetitefille o1)
2  {
3   Cmere o2;
4   Cfille o3;
5   ...
6   o3=o2;
7   f2(o3);
8  }
9
10 void f2(Cpetitefille o4)
11 {
12  ...
13  o4.g=10;
14  ...
15 }
```

Ces fonctions sont-elles correctes, *i.e.* est-ce qu'elles compilent ? Expliquez pourquoi.

CHAPITRE 8

Exercice 15. La surcharge de fonctions

Considérons deux classes nommées `Cmere` et `Cfille` (cette dernière dérive de `Cmere`). On suppose, par ailleurs, que la classe `Cfille` possède un constructeur, public, prenant un argument de type `char`. On dispose de trois versions de la fonction `f` dont les interfaces sont décrites ci-dessous.

```
1  int f(Cmere MERP1, int iP2)
2  { // Version 1
3   ...
4  }
5
6  int f(Cmere MERP1, double dP2)
7  { // Version 2
8   ...
9  }
10
11 double f(Cfille MERP1, int iP2)
12 { // Version 3
13  ...
14 }
```

Pour chacun des appels énumérés ci-dessous, expliquez quelle version sera appelée. On suppose que `fille` est un objet de type `Cfille`, `mere` un objet de type `Cmere`, `i` un `int`, `l` un `float`, `d` un `double` et `c` un `char`.

(1) `int res=f(mere,i);`

(2) `float res=f(mere,i);`

(3) `double res=f(fille,l);`

(4) `int res=f(fille,i);`

(5) `double res=f(fille,c);`

(6) `int res=f(i,l);`

Exercice 16. La surcharge d'opérateurs

Considérons la classe `Ccellule` que nous avons déjà vue dans l'exercice 12. On souhaite faire en sorte que le code ci-dessous soit correct.

```
1 Ccellule CELV1;
2 ...
3 if (CELV1[i].CELlire_element()==...) ...
4 ...
```

Quel opérateur faut-il surcharger ? Comment allez-vous écrire la surcharge ?

Exercice 17. La surcharge de types et les conversions

Dans cet exercice nous allons nous focaliser sur les conversions et la surcharge de types. Considérez dans un premier temps la classe `Ctest` déclarée ci-dessous.

```
1  class Ctest
2  {
3   private:
4    int iA;
5    float fB;
6    char *cLigne;
7   public:
8    Ctest();
9    ~Ctest();
10 };
```

(1) Déclarez et définissez la surcharge du type `int` de sorte que l'attribut `iA` soit retourné par cette surcharge.

(2) Comment faire pour que l'affectation `Ctest objet=4;` compile ? Écrire le code nécessaire.

(3) Expliquez le travail réalisé par le compilateur pour traiter les lignes suivantes.

```
1 Ctest objet1 ;
2 Ctest objet=4+objet1 ;
```

(4) Que se serait-il passé à la question 3 si on avait surchargé l'opérateur + dans la classe `Ctest` pour permettre l'addition avec un entier ?

(5) Expliquez le travail réalisé par le compilateur pour traiter les lignes suivantes.

```
1 Ctest objet1,objet ;
2 objet=4+objet1 ;
```

SOLUTIONS DES EXERCICES

Exercice 1. Déclaration et initialisation des variables

Le code proposé ne compile pas et cela à cause de la ligne 10 sur laquelle le programme fait référence à la variable `iBoucle2`. En effet, la portée de cette variable est limitée aux lignes 6 à 9, c'est-à-dire le bloc contenant sa déclaration.

Pour permettre à la ligne 10 de compiler, il faut déclarer la variable `iBoucle2` sur la ligne 4.

Exercice 2. Portée et visibilité des variables

À première vue on pourrait penser que la fonction `main` ne réalise qu'une seule fois la première boucle `for` puisque la variable `iBoucle` est redéfinie à l'intérieur du bloc correspondant avec une valeur supérieure à 3. Néanmoins, souvenez-vous que la portée de la variable `iBoucle` créée sur la ligne 7 est limitée aux lignes 6 à 10. Celle-ci masque la variable `iBoucle` définie sur la ligne 5 mais ne la remplace pas. Ainsi, la variable `iBoucle` créée sur la ligne 5 prendra les valeurs 0, 1 puis 2 avant de satisfaire la condition d'arrêt du `for`. L'affichage sera donc le suivant.

```
iBoucle (1) = 5
iBoucle (1) = 6
iBoucle (2) = 7
iBoucle (1) = 5
iBoucle (1) = 6
iBoucle (2) = 7
iBoucle (1) = 5
iBoucle (1) = 6
iBoucle (2) = 7
```

Exercice 3. Portée et visibilité des variables statiques

Par rapport à l'exercice précédent, l'affichage à l'écran diffère légèrement.

```
iBoucle (1) = 5
iBoucle (1) = 6
iBoucle (2) = 7
iBoucle (2) = 7
iBoucle (2) = 7
```

Pour comprendre ce résultat il faut vous remémorez deux points :

– La déclaration de la variable `iBoucle` sur la ligne 7 est équivalente à une déclaration à l'intérieur du second bloc `for` (donc sur la ligne 8).

– Le mot-clef `static` utilisé sur la ligne 7 indique au compilateur qu'il ne doit créer la variable `iBoucle` qu'une seule fois pour toutes les exécutions du second

bloc `for`. Ainsi, après la première exécution de ce bloc la variable `iBoucle` créée sur la ligne 7 vaudra toujours 7 et ne sera pas détruite en sortie de bloc. Donc, à la seconde exécution de ce bloc la condition d'arrêt sera déjà vérifiée ce qui provoquera un saut direct de la ligne 7 vers la ligne 9. Il en va de même lors de la troisième exécution du bloc.

Exercice 4. Passage d'arguments par référence

Voici le code de la fonction `f`.

```
1  void f(int & iP1)
2  {
3   iP1=10;
4  }
```

Exercice 5. Retour d'arguments par référence

Voici le code des fonctions `f` et `main`.

```
1  int & f()
2  {
3   int iV1=10;
4
5   return iV1;
6  }
7
8  void main()
9  {
10  int iV2;
11
12  iV2=f();
13 }
```

Ce code n'est pas judicieux car le mécanisme du passage par référence implique que ce qui est retourné par la fonction `f` est l'adresse sur la variable automatique `iV1`. Or la portée de cette variable est limitée à la fonction `f` donc, dans la fonction `main`, celle-ci n'existe plus et a été détruite sur la ligne 6. En conséquence, n'importe quelle valeur a pu être affectée à la variable `iV2`... tout dépend si le système a réutilisé, entre l'exécution des lignes 5 et 12, l'espace mémoire qui avait été alloué à la variable `iV1`.

Exercice 6. Passage d'arguments par référence : appel de fonction

– Ligne 12. Cette ligne compile car tous les arguments sont passés comme attendus par la fonction `f`. Notez que `&fV3` fait référence à *l'adresse de* `fV3` puisque celui-ci est déclaré (ligne 10) comme un objet et non pas un pointeur.

– Ligne 13. Cette ligne ne compile pas car le premier argument passé `&fV1` est une adresse alors que la fonction `f` attend un objet dont elle doit elle-même récupérer l'adresse (principe du passage par référence). Il y a donc incompatibilité : `f` attendait un objet et vous lui avez transmis une adresse. Pour que cette ligne compile il faut simplement se ramener à la syntaxe de la ligne 12.

– Ligne 14. Cette ligne ne compile pas à cause du troisième argument passé à la fonction `f`. En effet, l'argument `fV3` fait référence à un objet alors que la fonction `f` attendait une adresse. En conséquence, il faut passer `&fV3` au lieu de `fV3` pour que cela compile.

– Ligne 15. Cette ligne ne compile pas à cause du premier argument. En effet, vous passez à la fonction `f` un argument de type constant (la valeur 4) en guise d'objet dont elle doit retrouver l'adresse. Or, la valeur 4 ne possède pas d'adresse en mémoire ! D'où l'erreur de compilation. Pour remédier à cela vous devez indiquer dans l'en-tête de la fonction `f` que le premier argument peut être de type constant (ce qui implique que dans ce cas le compilateur ne passe pas l'objet constant par référence mais en fait une copie). Pour que la ligne 15 compile il faut donc que l'en-tête de la fonction `f` soit la suivante :

```
void f(const float &fV1, int iP2, float *fP3)
```

Exercice 7. Variables références

Voici le code de la fonction `f`.

```
1 void f()
2 {
3  int iV1;
4  int &riV2 = iV1;
5 }
```

Exercice 8. Déclarer, définir et utiliser des classes : la classe `Csac`

(1) Voici l'interface de la classe `Csac`, contenue dans le fichier `exercice8.h`. Notez ici la présence des spécifications des méthodes et les conséquences de la non prise en compte des exceptions (toutes les conditions de fonctionnement sont fixées en précondition).

```
1
2 #define TYPSacPlastique 1
3 #define TYPSacADos 2
4 #define TYPSacSport 3
5
6 class Csac
7 {
8  // Liste des attributs
```

```
9   private :
10          int iSACType; // Type de sac
11          int iSACNbElements; // Nombre d'éléments dans le sac
12          int iSACListeElements[50]; // Liste des éléments dans le sac
13                                     // (50 au maximum)
14
15  // Liste des primitives
16  public :
17          Csac();
18      /* Constructeur par défaut de la classe
19           E : néant
20           nécessite : néant
21           S : néant
22           entraîne : Le sac est vide et sans type */
23          void SACFixerType(const int & iP1);
24      /* Permet de fixer le type de sac
25           E : iP1, le type du sac
26           nécessite : iP1 correspond α un type existant
27           S : néant
28           entraîne : Le type du sac est modifié */
29          int SACLireType();
30      /* Permet de connaître le type de sac
31           E : néant
32           nécessite : néant
33           S : Le type du sac
34           entraîne : Le type du sac est renvoyé */
35          int SACTaille();
36      /* Permet de connaître la taille du sac
37           E : néant
38           nécessite : néant
39           S : La taille du sac
40           entraîne : La taille du sac est renvoyée */
41          int SACEnleverElement();
42      /* Permet de retirer le dernier élément du sac
43           E : néant
44           nécessite : Le sac n'est pas vide
45           S : L'élément retiré
46           entraîne : Le dernier élément du sac est retiré */
47          void SACAjouterElement(const int & iP1);
48      /* Permet d'ajouter un élément dans le sac
49           E : iP1, l'élément α ajouter
50           nécessite : Le sac n'est pas plein
51           S : néant
52           entraîne : L'élément est ajouté α la fin du sac */
53  };
```

(2) Voici le corps de la classe `Csac`, contenue dans le fichier `exercice8.cpp`.

```
1  #include "exercice8.h"
2
3  Csac : :Csac()
4  {
5   iSACType=0;
6   iSACNbElements=0;
7   // Le sac est vide
8  }
9
10 void Csac : :SACFixerType(const int & iP1)
11 {
12  iSACType=iP1;
13  // Le type du sac est changé
14 }
15
16 int Csac : :SACLireType()
17 {
18  return(iSACType);
19  // Le type du sac est renvoyé
20 }
21
22 int Csac : :SACTaille()
23 {
24  return(iSACNbElements);
25  // Le nombre d'éléments dans le sac est renvoyé
26 }
27
28 int Csac : :SACEnleverElement()
29 {
30  iSACNbElements--;
31  return(iSACListeElements[iSACNbElements]);
32  // Le dernier élément du sac est retiré et retourné
33 }
34
35 void Csac : :SACAjouterElement(const int & iP1)
36 {
37  iSACListeElements[iSACNbElements]=iP1;
38  iSACNbElements++;
39  // L'élément est ajouté α la fin du sac
40 }
```

(3) Voici la fonction `main` demandée. Vous noterez bien que dans le sac `SACV2` l'ordre des éléments est inversé par rapport à ceux rangés dans le sac `SACV1`.

```
1  #include "exercice8.h"
2
3  void main()
4  {
5   Csac SACV1, SACV2;
6   // Les objets sont initialisés par appel au constructeur par défaut de la classe
7   int iBoucle;
8
9   SACV1.SACFixerType(TYPSacADos);
10  SACV2.SACFixerType(TYPSacADos);
11  // Les types de sac sont fixés
12
13  for (iBoucle=1;iBoucle<=10;iBoucle++)
14          SACV1.SACAjouterElement(iBoucle);
15  // Les éléments de 1 α 10 sont ajoutés au sac
16
17  for (iBoucle=1;iBoucle<=10;iBoucle++)
18          SACV2.SACAjouterElement(SACV1.SACEnleverElement());
19  // Les éléments sont transférés du premier sac vers le second
20  }
```

(4) Pour cela il suffit de déclarer un attribut `static` au sein de la classe et de créer un accesseur pour la lecture de cet attribut. L'accesseur peut être une méthode statique, c'est au choix du développeur. Voici les modifications apportées au sein du fichier `exercice8.h`.

```
1  #define TYPSacPlastique 1
2  #define TYPSacADos 2
3  #define TYPSacSport 3
4
5  class Csac
6  {
7   // Liste des attributs
8   private :
9           int iSACType; // Type de sac
10          int iSACNbElements; // Nombre d'éléments dans le sac
11          int iSACListeElements[50]; // Liste des éléments dans le sac
12                                     // (50 au maximum)
13      static int iSACCompteur; // Nombre de sacs créés dans le programme
14                                     // (50 au maximum)
15
16  // Liste des primitives
17  public :
18          int SACLireCompteur();
```

```
19      /* Permet de connaître le nombre de sacs créés
20            E : néant
21            nécessite : néant
22            S : Le nombre de sac
23            entraîne : Le nombre de sacs créés est retourné */
24          Csac() ;
25      /* Constructeur par défaut de la classe
26            E : néant
27            nécessite : néant
28            S : néant
29            entraîne : Le sac est vide et sans type */
30
31 ...
```

Voici maintenant les modifications apportées au fichier `exercice8.cpp`.

```
1  #include "exercice8.h"
2
3  int Csac : :iSACCompteur=0 ;
4
5  int Csac : :SACLireCompteur()
6  {
7   return(iSACCompteur) ;
8   // Le nombre de sacs créés est retourné
9  }
10
11 Csac : :Csac()
12 {
13  iSACType=0 ;
14  iSACNbElements=0 ;
15  iSACCompteur++ ;
16  // Le sac est vide
17 }
18
19 ...
```

Exercice 9. Déclarer, définir et utiliser des classes : la classe `CListe`

(1) L'interface de la classe `Cliste` (fichier `Cliste.h`) correspondant aux besoins énoncés pour la classe `Csac` de l'exercice précédent est donnée ci-dessous.

```
1  typedef int Telement ; // Type d'un élément de liste, par défaut int
2
3  class Cliste
4  {
```

```
5   // Liste des attributs
6   private :
7         unsigned int uiTaille ; // Nombre d'éléments dans la liste
8         Telement *Tliste ; // Liste des éléments
9
10  // Liste des primitives
11  public :
12    // Les constructeurs et destructeurs
13    Cliste() ;
14    Cliste(Cliste & LISP1) ;
15    ~Cliste() ;
16
17    // Les méthodes
18    void LISajouter_element(Telement TP1) ; // Permet d'ajouter un élément à la fin
19    Telement LISenlever_element() ; // Retire le dernier élément
20    unsigned int LIStaille() ; // Renvoie le nombre d'éléments dans la liste
21    Telement LISlire_element(unsigned int uiPos) ; // Permet de lire un élément
22              // à une position donnée
23  } ;
```

Cette interface est minimaliste. Il faut noter qu'il est également nécessaire de prévoir une surcharge de l'opérateur d'affectation puisque nous avons des attributs dynamiques dans la classe. Par ailleurs, pour bien faire il faudrait également intégrer une gestion des exceptions notamment à cause des problèmes pouvant survenir lors de l'allocation/réallocation des pointeurs.

(2) Le corps de la classe `Cliste` (fichier `Cliste.cpp`) est donné ci-dessous.

```
1   #include "exercice9.h"
2   #include<stdlib.h>
3   #include<malloc.h>
4
5   Cliste : :Cliste()
6   { // Constructeur par défaut
7     uiTaille = 0 ;
8     Tliste = NULL ;
9     // La liste est vide
10  }
11
12  Cliste : :Cliste(Cliste & LISP1)
13  { // Constructeur de recopie
14    unsigned int iBoucle ;
15
16    uiTaille = LISP1.uiTaille ;
17    Tliste = (Telement *)malloc(uiTaille*sizeof(Telement)) ;
18    for (iBoucle=0 ;iBoucle<uiTaille ;iBoucle++)
```

```
19              Tliste[iBoucle]=LISP1.Tliste[iBoucle];
20   // L'objet est initialisé par recopie
21  }
22
23  Cliste : :~Cliste()
24  { // Destructeur
25   free(Tliste);
26   // L'objet est prêt à être détruit
27  }
28
29  void Cliste : :LISajouter_element(Telement TP1)
30  { // Permet d'ajouter un élément à la fin de la liste
31   Tliste=(Telement *)realloc(Tliste,(uiTaille+1)*sizeof(Telement));
32   Tliste[uiTaille]=TP1;
33   uiTaille++;
34   // L'élément est ajouté à la fin s'il y a assez de mémoire
35  }
36
37  Telement Cliste : :LISenlever_element()
38  { // Permet de retirer un élément à la fin de la liste
39   Telement TV1;
40
41   TV1=Tliste[uiTaille-1];
42   Tliste=(Telement *)realloc(Tliste,(uiTaille-1)*sizeof(Telement));
43   uiTaille--;
44   // L'élément est retiré de la fin de la liste
45   return(TV1);
46  }
47
48  unsigned int Cliste : :LIStaille()
49  { // Renvoie le nombre d'éléments dans la liste
50   return(uiTaille);
51  }
52
53  Telement Cliste : :LISlire_element(unsigned int uiPos)
54  { // Permet de lire un élément si l'élément est dans le tableau
55   if (uiPos<uiTaille) return(Tliste[uiPos]);
56              else exit(1);
57  }
```

Exercice 10. Allocation d'objets dynamiques

Voici le détail de ce que fait chacune des fonctions.

– *Fonction `f1`*. Cette fonction crée un pointeur de type `Cliste` (ligne 3) que l'on fait pointer sur un objet alloué mais non initialisé (ligne 5).

– *Fonction `f2`*. Cette fonction crée un pointeur de type `Cliste` (ligne 11) que l'on fait pointer sur un objet alloué et initialisé par appel au constructeur par défaut de la classe `Cliste` (ligne 13).
– *Fonction `f3`*. Cette fonction crée un pointeur de type `Cliste` (ligne 19) que l'on fait pointer sur une liste de 10 objets alloués mais non initialisés (ligne 21).
– *Fonction `f4`*. Cette fonction crée un pointeur de type `Cliste` (ligne 27) que l'on fait pointer sur une liste de 10 objets alloués et initialisés par appel au constructeur par défaut de la classe `Cliste` (ligne 29). Chacun des 10 objets fait l'appel d'un appel à ce constructeur.

On n'utilisera jamais les version `f1` et `f3` auxquelles on préfèrera systématiquement les versions `f2` et `f4`.

Exercice 11. Allocation d'objets dynamiques : les doubles pointeurs

(1) La fonction `f5` :
  – Crée un double pointeur de type `Cliste` (ligne 4),
  – Fait pointer le premier niveau du pointeur sur une liste de 10 adresses de type `Cliste *` (ligne 6),
  – Fait pointeur chacune de ces 10 adresses (second niveau du pointeur) sur une liste de 20 objets de type `Cliste` initialisés par appel au constructeur par défaut pour chacun (ligne 7).

  Autrement dit, vous venez de créer une matrice (aussi appelée tableau à double entrée) de 10x20 objets du type `Cliste`. Cette fonction est correcte puisque la fonction `malloc` n'a été utilisée que sur des adresses et pas sur des objets.
  Vous avez ici un exemple qui vous indique comment procéder quand vous voulez créer des listes d'objets de taille dynamique ! Vous pouvez alors faire un `realloc` sur le premier niveau de pointeur, sachant qu'à chaque adresse vous allez faire pointer non pas sur 20 objets mais sur 1 seul.

(2) Cette version de la fonction provoque le même résultat que la version de la question 1. Il n'y a aucune différence ici dans l'utilisation des fonctions `malloc` et `new`.

Exercice 12. Déclarer, définir et utiliser des classes : la classe `Ccellule`

(1) Voici l'interface de la classe `Ccellule`. Notez bien l'absence de la surcharge de l'opérateur d'affectation étant donné que la surcharge est abordée dans un chapitre ultérieur. Néanmoins, dans une implémentation utilisée de cette classe la surcharge est nécessaire.

  Par ailleurs, la liste des méthodes proposée ici est ce qu'il est nécessaire d'avoir pour pouvoir faire fonctionner la classe.

```
1  class Ccellule
2  {
3   // Liste des attributs
4   private :
5        Ccellule * CELPrec ; // Prédécesseur dans la liste chaînée
6        int iCELelement ; // L'élément contenu
7        Ccellule * CELSuiv ; // Prédécesseur dans la liste chaînée
8
9   // Liste des primitives
10  public :
11   // Les constructeurs et destructeurs
12   Ccellule() ;
13   Ccellule(Ccellule & CELP1) ;
14   ~Ccellule() ;
15
16   // Les méthodes
17   void CELchainer(Ccellule * CELP1) ; // Permet de chaîner la cellule en cours
18              // après la cellule passée en paramètre
19   void CELdechainer() ; // Retire la cellule du chaînage
20   int CELlire_element() ; // Retourne la valeur de l'élément
21   void CELmodifier_element(int iV1) ; // Permet de modifier l'élément
22   Ccellule * CELchercher_prec(int iPos) ; // Retourne la iPos-ème
23              // cellule précédente
24   Ccellule * CELchercher_suiv(int iPos) ; // Retourne la iPos-ème
25              // cellule suivante
26 } ;
```

(2) Étant donnée l'interface de la réponse à la question précédente, l'exécution du code va conduire au résultat suivant :

- Ligne 3. Création d'un objet, nommé `cellule1`, de la classe `Ccellule` initialisé par appel au constructeur par défaut.
- Ligne 5. Comme du code a été exécuté sur la ligne 4 on suppose, sans perte de généralité, que l'objet `cellule1` est inséré dans un chaînage (il possède donc `CELPrec` ou `CELSuiv` non `NULL`). Sur la ligne 5, il y a création d'un objet, nommé `cellule2`, de la classe `Ccellule` initialisé par appel au constructeur de recopie à partir de `cellule1`. **D'après vous que fait le constructeur de recopie que vous avez déclaré dans l'interface ?** Et bien, il recopie juste la valeur de `CELelement` et c'est tout ! Pas question de recopier les pointeurs sous peine d'avoir des courts-circuits dans le chaînage : `cellule1` et `cellule2` auraient les mêmes cellules précédentes et suivantes.
- Ligne 7. Elle est similaire à la ligne 3.
- Ligne 8. Sur cette ligne nous réalisons l'affectation d'un objet de la classe `Ccellule` dans un autre objet de la même classe. Le compilateur va donc

utiliser l'opérateur d'affectation par défaut qui recopie membre à membre. Nous sommes donc en train d'introduire un court-circuit dans le chaînage puisqu'après la recopie membre à membre les objets `cellule3` et `cellule2` auront les mêmes prédécesseurs et successeurs. Cela va conduire potentiellement à des plantages dans la suite du code.

(3) Pour résoudre le problème soulevé précédemment il suffit de définir sa propre version de l'opérateur d'affectation pour éviter l'appel à la version par défaut et donc la recopie membre à membre. Bref, il suffit de suivre les consignes données dans le cours !

(4) Voici la définition de la méthode `CELchainer`.

```
1  void Ccellule : :CELchainer(Ccellule * CELP1)
2  { // Le paramètre CELP1 ne doit pas être NULL
3   Ccellule *CELtmp;
4
5   CELtmp=CELP1->CELSuiv;
6   CELP1->CELSuiv=this;
7   CELPrec=CELP1;
8   CELSuiv=CELtmp;
9   if (CELSuiv!=NULL) CELSuiv->CELPrec=this;
10 };
```

Exercice 13. L'héritage simple et les exceptions

(1) Voici l'interface de la classe `Ccellule` (fichier `Ccellule.h`).

```
1  // Valeurs pour les exceptions levées
2  #define Predecesseur_inexistant 100
3  #define Successeur_inexistant 101
4
5  class Ccellule
6  { // 2nde version de la classe Ccellule
7   // Liste des attributs
8   protected :
9            Ccellule * CELPrec; // Prédécesseur dans la liste chaînée
10           Ccellule * CELSuiv; // Prédécesseur dans la liste chaînée
11
12  // Liste des primitives
13  public :
14   // Les constructeurs et destructeurs
15   Ccellule();
16   Ccellule(Ccellule & CELP1);
17   ~Ccellule();
18
```

```
19    // Les méthodes
20    void CELchainer(Ccellule * CELP1); // Permet de chaîner la cellule en cours
21              // après la cellule passée en paramètre
22    void CELdechainer(); // Retire la cellule du chaînage
23    Ccellule * CELchercher_prec(int iPos); // Retourne la iPos-ème
24              // cellule précédente
25              // Cette fonction peut lever l'exception Predecesseur_inexistant
26    Ccellule * CELchercher_suiv(int iPos); // Retourne la iPos-ème
27              // cellule suivante
28              // Cette fonction peut lever l'exception Successeur_inexistant
29  };
```

Notez ici la présence du contrôle d'accès `protected` afin de garantir que seules les classes héritées auront accès à ces deux attributs. Le corps de la classe est le suivant (fichier `Ccellule.cpp`).

```
1   #include "Ccellule.h"
2   #include "Cexception.h"
3   #include<stdlib.h>
4
5   Ccellule : :Ccellule()
6   { // Constructeur par défaut
7           CELPrec = NULL;
8           CELSuiv = NULL;
9    // L'objet est initialisé
10  }
11
12  Ccellule : :Ccellule(Ccellule & CELP1)
13  { // Constructeur de recopie
14          CELPrec = NULL;
15          CELSuiv = NULL;
16    // L'objet est initialisé : les pointeurs de l'objet passé en
17    // paramètre ne peuvent pas être recopiés sous peine de créer
18    // un court-circuit dans le chaînage.
19  }
20
21  Ccellule : :~Ccellule()
22  { // Destructeur
23  }
24
25  void Ccellule : :CELchainer(Ccellule * CELP1)
26  { // Cette méthode insère une cellule après le paramètre CELP1 dans le chaînage
27    // Le paramètre CELP1 ne doit pas être NULL
28    // La cellule en cours (this) ne doit pas être déjà chaînée
29   Ccellule *CELtmp;
30
```

```
31   CELtmp=CELP1->CELSuiv;
32   CELP1->CELSuiv=this;
33   CELPrec=CELP1;
34   CELSuiv=CELtmp;
35   if (CELSuiv!=NULL) CELSuiv->CELPrec=this;
36 };
37
38 void Ccellule : :CELdechainer()
39 { // Cette méthode permet de retirer la cellule en cours du chaînage
40         if (CELPrec != NULL) CELPrec->CELSuiv = CELSuiv;
41         if (CELSuiv != NULL) CELSuiv->CELPrec = CELPrec;
42         CELPrec = NULL;
43         CELSuiv = NULL;
44 }
45
46 Ccellule * Ccellule : :CELchercher_prec(int iPos)
47 { // Cette méthode permet de rechercher un élément à une position précédente
48         Ccellule * CELEnCours;
49         int iBoucle=0;
50
51         CELEnCours = this;
52         while(iBoucle<iPos)
53         {
54                 CELEnCours = CELEnCours->CELPrec;
55                 iBoucle++;
56                 if (CELEnCours==NULL)
57                 { // On lève une exception : le prédécesseur n'existe pas
58                   Cexception EXCerreur;
59                   EXCerreur.EXCmodifier_valeur(Predecesseur_inexistant);
60                   throw EXCerreur;
61                 }
62         }
63         return CELEnCours;
64 }
65
66 Ccellule * Ccellule : :CELchercher_suiv(int iPos)
67 { // Cette méthode permet de rechercher un élément à une position suivante
68         Ccellule * CELEnCours;
69         int iBoucle=0;
70
71         CELEnCours = this;
72         while(iBoucle<iPos)
73         {
74                 CELEnCours = CELEnCours->CELSuiv;
75                 iBoucle++;
76                 if (CELEnCours==NULL)
```

```
77                  { // On lève une exception : le prédécesseur n'existe pas
78                    Cexception EXCerreur;
79                    EXCerreur.EXCmodifier_valeur(Successeur_inexistant);
80                    throw EXCerreur;
81                  }
82          }
83          return CELEnCours;
84  }
```

Un point important est lié à la présence du constructeur de recopie qui ne fait que mettre les pointeurs à `NULL`. Cela est nécessaire puisque sinon nous introduirions des courts-circuits dans la liste chaînée. Cela à d'autant plus d'importance dans le cadre de la seconde question.

Enfin, regardez bien les lignes 56-61 et 76-81 sur lesquelles on fait remonter une exception car l'élément recherché n'existe pas. Les valeurs passées à l'exception sont définies dans l'interface de la classe : c'est ainsi que vous informerez votre utilisateur de la classe `Ccellule` de la levée d'exceptions et quelles exceptions peuvent être levées.

Il est important de remarquer ici qu'il manque la surcharge de l'opérateur d'affectation pour que la correction soit complètement correcte.

(2) Voici l'interface de la classe `Cliste_entier` (fichier `Cliste_entier.h`).

```
1   #include "Ccellule.h"
2
3   // Valeurs pour les exceptions levées
4   #define Element_inexistant 200
5
6   class Cliste_entier : private Ccellule
7   {
8    private :
9      int iElement; // L'élément d'une cellule de la liste
10
11   // Liste des primitives
12   public :
13    // Les constructeurs et destructeurs
14    Cliste_entier();
15
16    // Les méthodes
17   int LIElire_element(int iPos); // Cette méthode permet de connaître
18                     // le iPos-ème élément qui suit dans la liste
19                     // Cette méthode peut lever l'exception Element_inexistant si iPos
20                     // est trop grand.
21   void LIEmodifier_element(int iPos, int iVal);
22                     // Cette méthode affecte la valeur iVal à l'élément situé dans la
```

```
23                  // position iPos.
24                  // Cette méthode peut lever l'exception Element_inexistant si iPos
25                  // est trop grand.
26   void LIEajouter_element(int iPos, int iVal);
27                  // Cette méthode ajoute la valeur iVal dans la
28                  // position iPos.
29                  // Cette méthode peut lever l'exception Element_inexistant si iPos
30                  // est trop grand.
31   int LIEsupprimer_element(int iPos); // Cette méthode permet de supprimer
32                  // le iPos-ème élément qui suit dans la liste
33                  // Cette méthode peut lever l'exception Element_inexistant si iPos
34                  // est trop grand.
35 };
```

Il est important de noter ici que l'héritage est privé. Pourquoi ? Et bien tout simplement parce que l'utilisateur de la classe `Cliste_entier` n'a pas besoin d'utiliser les méthodes de la classe `Ccellule`. On réalise ainsi *une encapsulation de la classe mère*. L'utilisateur de la classe `Cliste_entier` n'utilise donc que les primitives liées à la gestion d'une liste : ajouter un élément, lire un élément... D'un point de vue génie logiciel, cela est très bien.

Analysons maintenant les constructeurs. Seul le constructeur par défaut, dans la classe `Cliste_entier` a été prévu (et encore, il n'est pas vraiment nécessaire). En effet, si vous appliquez la règle vue dans la partie cours, vous n'avez pas besoin de prévoir de constructeur de recopie (ni de surcharge de l'opérateur d'affectation) puisqu'il n'y a pas d'attributs dynamiques au sein de la classe (les attributs dynamiques sont hérités). En pratique, cela encore est parfait : lorsque vous initialiserez par recopie un objet de la classe `Cliste_entier`, le constructeur de recopie par défaut sera appelé pour l'attribut de cette classe et il y aura appel au constructeur de recopie défini dans la classe `Ccellule` pour les attributs hérités de cette classe. Le comportement résultant est celui attendu. De même, il n'est nul besoin de destructeur.

Maintenant examinons les exceptions. Toutes les méthodes de la classe `Cliste_entier` peuvent lever la même exception qui est en fait l'exception `Successeur_inexistant` qui est relayée car aucune de ces méthodes ne sait gérer le cas où on cherche à manipuler un élément trop loin dans la liste.

(3) La réponse à cette question est contenue dans la réponse à la question précédente. En effet, dans la classe `Ccellule`, les attributs `CELPrec` et `CELSuiv` sont définis en `protected`. Si on fait un héritage autre que `private` alors ces attributs vont garder le même statut dans la classe `Cliste_entier` et seront donc accessibles à nouveau par héritage dans les classes filles de `Cliste_entier`. Ainsi, pour éviter cela il suffit de faire un héritage privé comme indiqué dans la réponse à la question 2. Ces deux attributs sont alors `private` dans la classe `Cliste_entier` et inacessibles lors d'un autre héritage.

Exercice 14. L'héritage simple

(1) Ces trois classes compilent mais provoquent un comportement "anormal". En effet, le constructeur et destructeur de la classe `Cmere` étant déclarés en `protected`, toute déclaration d'un objet de cette provoquera une erreur de compilation puisque vous n'aurez pas accès au constructeur. Il en va de même si vous déclarez des objets des classes dérivées.

Par contre, la déclaration de la ligne 27 n'est pas une erreur : la classe `Cpetitefille` possède deux attributs `a` dont un lui vient par héritage de la classe `Cmere`.

Pour s'assurer que ces classes sont utilisables il suffit de rendre public les constructeurs.

(2) Voici le tableau récapitulatif des statuts, accès interne et accès externe pour chaque attribut des trois classes.

Attribut	Classe `Cmere`			Classe `Cfille`		
	Statut initial	Accès interne	Accès externe	Statut	Accès interne	Accès externe
`a`	`public`	Oui	Oui	`private`	Oui	Non
`b`	`public`	Oui	Oui	`private`	Oui	Non
`c`	`private`	Oui	Non	`private`	Non	Non
`d`		——		`public`	Oui	Oui
`e`		——		`protected`	Oui	Non

Attribut	Classe `Cpetitefille`		
	Statut	Accès interne	Accès externe
`Cmere::a`	`private`	Non	Non
`b`	`private`	Non	Non
`c`	`private`	Non	Non
`d`	`public`	Oui	Oui
`e`	`protected`	Oui	Non
`a`	`public`	Oui	Oui
`g`	`public`	Oui	Oui

Notez bien que l'héritage entre les classes `Cmere` et `Cfille` est privé puisqu'aucun contrôle d'accès n'est précisé.

(3) L'attribut `d` qui est référencé est celui de la classe `Cfille`. Pour l'attribut `a` il s'agit de celui de la classe `Cpetitefille`. En effet, la méthode `f` appartenant à cette classe, le compilateur va d'abord chercher à l'intérieur de la classe avant de chercher dans les ascendants. On dit alors que l'attribut `a` de la classe `Cpetitefille` *occulte* l'attribut `a` de la classe `Cmere`. En regardant le tableau récapitulatif de la question précédente on s'aperçoit que l'affectation est possible car on a accès (accès interne) à ces deux attributs.

Pour référencer l'autre attribut `a` il faudrait le nommer explicitement, ce qui évitera au compilateur de *chercher*. On utilise donc le nom long et l'affectation devient : `Cmere::a=d;`

Malheureusement cette affectation ne compile plus car on n'a pas d'accès interne à l'attribut `Cmere::a` dans la méthode `f`.

(4) Ces fonctions ne compilent pas. Rappelez-vous la règle vue dans le chapitre 7, section 7.1.4. : *Là où la mère va la fille va, là où la fille passe la mère trépasse !* Reprenons la ligne 6 qui pose problème. La réalisation de l'affectation nécessite de mettre un objet de la classe `Cmere` dans un objet de la classe `Cfille`. Autrement dit, l'opérateur d'affectation[3] attend à droite du signe "=" un objet de la classe `Cfille`, or nous lui avons mis un `Cmere` : *là où la fille passe la mère trépasse !* Cette ligne ne peut donc pas compiler. Une autre façon de voir les choses consiste à simuler à la main la recopie membre à membre qui serait mise en œuvre par le compilateur pour réaliser l'opération : quelles seraient les valeurs prises pour initialiser les attributs `d` et `e` de la classe `Cfille` puisque la `Cmere` ne les possède pas ?

De la même façon, la ligne 7 ne peut pas compiler puisque nous lui passons un objet de la classe `Cfille` alors que la fonction attend un objet de la classe `Cpetitefille`, donc avec plus d'attributs. N'oubliez pas de plus que comme nous réalisons un passage par valeur à la fonction `f2`, l'objet `o4` cité en paramètre est un objet temporaire initialisé par appel au constructeur de recopie. Autrement dit, la ligne 7 provoque l'appel au constructeur de recopie de la classe `Cpetitefille` en passant en paramètre à celui-ci un objet de la classe `Cfille` : *là où la petite-fille passe la fille trépasse !*

Exercice 15. La surcharge de fonctions

Voici les réponses en fonction des appels.

– "`int res=f(mere,i);`". Ici c'est la version 1 qui est appelée au titre de la *correspondance exacte* pour tous les arguments. En effet, tous les types des objets et variables passés lors de l'appel à `f` correspondent à ceux attendus dans la version 1.

– "`float res=f(mere,i);`". La réponse est ici identique au cas précédent : version 1. N'oubliez pas que le type de retour n'a aucune incidence sur le choix de la version appelée : la version 1 retourne un `int` et `res` est de type `float`... et bien, après appel à `f` le compilateur mettra en place une conversion de `int` vers `float`.

– "`double res=f(fille,l);`". Le compilateur vous met un message d'erreur sur cette ligne de type "Ambiguïté" : les trois versions peuvent être appelées. L'application du pseudo-algorithme donné dans la section 8.1.3. donne : $E_1 = \{version1, version2, version3\}$, $E_2 = \{version1, version2, version3\}$ et $E = \{version1, version2, version3\}$. Il n'y a aucune version dominée dans E et aucune fonction ne peut être choisie uniquement au titre de la *correspondance*

3. On suppose évidemment qu'il n'est pas défini, au sein de la classe `Cfille`, de surcharge de cet opérateur prenant en paramètre un objet de la classe `Cmere`. Dans le cas contraire cette ligne compilerait et ferait appel à cette surcharge.

exacte ou de la *promotion numérique* pour tous les arguments. Conclusion : les trois versions peuvent être appelées.

– "`int res=f(fille,i);`". Au titre de la *correspondance exacte* pour tous les arguments, c'est la version 3 qui va être appelée.

– "`double res=f(fille,c);`". Ici il n'existe pas de version qui corresponde exactement pour tous les arguments. L'application du pseudo-algorithme vu dans la section 8.1.3. donne : $E_1 = \{version1,\ version2, version3\}$, $E_2 = \{version1, version2, version3\}$ et $E = \{\ version1, version2, version3\}$. Or dans E les versions 1 et 2 sont dominées par la version 3 car ces deux versions mettent en œuvre une conversion définie (sur le premier argument) tandis que la version 3 n'en fait aucune. Donc, $E = \{version3\}$ et le compilateur choisit la version 3.

– "`int res=f(i,l);`". Au premier abord on peut se dire que ce code ne compile pas. Et pourtant si. Évidemment, aucune des trois versions ne peut être appelée au titre de la correspondance exacte pour tous les arguments. L'application du pseudo-algorithme vu dans la section 8.1.3. donne : $E_1 = \{version3\}$, $E_2 = \{version1, version2, version3\}$ et $E = \{version3\}$. La version 3 peut être appelée pour le premier argument grâce à la chaîne de conversion suivante : `int` $\rightarrow$ `char` $\rightarrow$ `Cfille`. On a ainsi une chaîne à deux conversions dont une CDU[4] ce qui est réalisable. La CDU est effectuée par la création d'un objet temporaire de la classe `Cfille` initialisé par appel au constructeur à un argument de type `char`[5].

Exercice 16. La surcharge d'opérateurs

Il faut surcharger l'opérateur d'indexation [] comme suit.

```
1  class Ccellule
2  {
3   ...
4   public :
5    Ccellule & operator[](int iPos);
6  };
7  ...
8  Ccellule & Ccellule : :operator[](int iPos)
9  {
10  return (* CELchercher_suiv(iPos));
11 }
```

La surcharge réalisée ici retourne la cellule par référence, ce qui veut dire que les méthodes qui modifient l'objet (par exemple, `CELmodifier_element`) modifient

4. Conversion Définie par l'Utilisateur.
5. Ce constructeur aurait été `private` ou `protected`, l'appel n'aurait pas pu avoir lieu.

réellement la cellule et non pas une copie de celle-ci. Pour que cela ne puisse pas être le cas il aurait fallu faire un retour par valeur et non pas un retour par référence.

Exercice 17. La surcharge de types et les conversions

(1) La classe `Ctest` incluant la définition de la surcharge du type `int` est donnée ci-dessous.

```
1  class Ctest
2  {
3   private :
4    int iA ;
5    float fB ;
6    char *cLigne ;
7   public :
8    Ctest() ;
9    ~Ctest() {} ;
10   operator int() {return iA ;}
11 } ;
```

(2) Pour que cette initialisation lors d'une déclaration soit possible, il faut définir un constructeur à un argument de type `int`.

```
1  class Ctest
2  {
3   private :
4    int iA ;
5    float fB ;
6    char *cLigne ;
7   public :
8    Ctest() ;
9    Ctest(int iP1) {iA=iP1 ;}
10   ~Ctest() {} ;
11   operator int() {return iA ;}
12 } ;
```

(3) Voici le travail réalisé par le compilateur sur ces lignes.

- Ligne 1. Le compilateur crée un objet, nommé `objet1`, initialisé par appel au constructeur par défaut de la classe `Ctest`.
- Ligne 2. Le compilateur cherche à créer un objet, nommé `objet`, pour lequel il doit déterminer le constructeur à appeler. Pour cela il doit évaluer le résultat de l'addition. *Existe-t-il un opérateur + capable de faire l'addition d'un entier avec un `Ctest` ?* La réponse est non. Donc, le compilateur va chercher à mettre en œuvre des chaînes de conversion. Tout d'abord, il n'est pas intéressant pour lui de convertir 4 en un `Ctest` puisqu'il ne dispose pas d'une

version de l'opérateur + permettant de faire l'addition entre deux `Ctest`. Par contre, il peut réaliser la conversion de `objet1` en `int` grâce à la surcharge du type `int` dans la classe `Ctest`. Le résultat est donc un `int` qui est additionné avec 4. Le résultat de l'addition est donc un entier et l'objet `objet` sera initialisé par appel au constructeur à un argument de type entier.

Ces deux lignes compilent donc correctement.

(4) Voici le travail réalisé par le compilateur sur ces lignes.

- Ligne 1. Le compilateur crée deux objets, nommés `objet1` et `objet`, initialisés par appel au constructeur par défaut de la classe `Ctest`.
- Ligne 2. Le compilateur cherche à affecter dans `objet` le résultat de l'addition. *Existe-t-il un opérateur + capable de faire l'addition d'un entier avec un `Ctest` ?* La réponse est non. Donc, le compilateur va chercher à mettre en œuvre des chaînes de conversion. Tout d'abord, il n'est pas intéressant pour lui de convertir 4 en un `Ctest` puisqu'il ne dispose pas d'une version de l'opérateur + permettant de faire l'addition entre deux `Ctest`. Par contre, il peut réaliser la conversion de `objet1` en `int` grâce à la surcharge du type `int` dans la classe `Ctest`. Le résultat est donc un `int` qui est additionné avec 4.
  Le résultat de l'addition est donc un entier qui doit être affecté dans `objet`. *Existe-t-il un opérateur d'affectation, surchargé dans la classe `Ctest`, qui prenne en argument un objet de type `int` ?* La réponse est non. Donc le compilateur va chercher à convertir le résultat de l'addition en un objet de type `Ctest` : cette conversion est possible grâce à la création d'un objet temporaire initialisé par appel au constructeur à un argument de type `int`. Cet objet temporaire est alors recopié membre à membre dans `objet` par appel à l'opérateur d'affectation par défaut.

Ces deux lignes compilent donc correctement.

ANNEXES

Annexe **A**

Les mots-clefs du langage C++ et la construction d'identifiants

Les identifiants suivants sont des mots-clefs du langage C++. Ils reprennent les mots-clefs du langage C et ne doivent pas être redéfinis au-delà des limites imposées par le langage.

```
asm             false             sizeof
auto            float             static
bool            for               static_cast
break           friend            struct
case            goto              switch
catch           if                template
char            inline            this
class           int               throw
const           long              true
const_cast      mutable           try
continue        namespace         typedef
default         new               typeid
delete          operator          typename
do              private           union
double          protected         unsigned
dynamic_cast    public            using
else            register          virtual
enum            reinterpret_cast  void
explicit        return            volatile
export          short             wchar_t
extern          signed            while
```

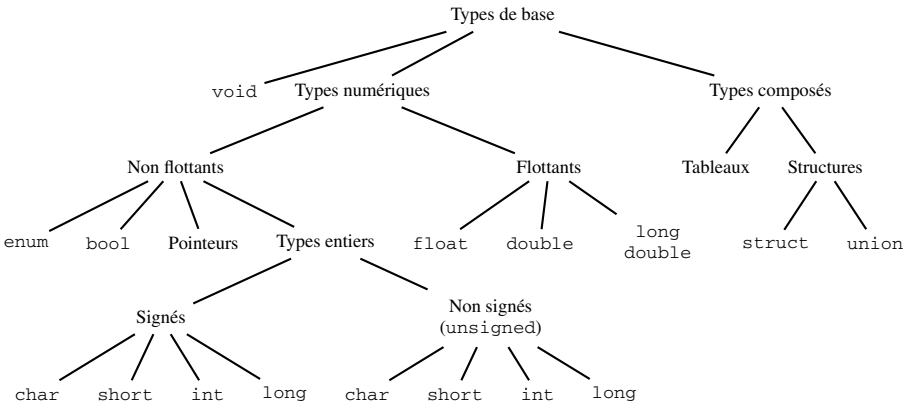
Lorsque vous programmez vous êtes amenés à définir des noms de fonctions, de classes, de variables... bref à définir vos propres identifiants. En règle générale, il n'est pas possible de définir un identifiant déjà présent dans le langage C++ (ou alors de façon très particulière et selon des règles bien définies, comme dans le cas de la surcharge de types). Les règles que vous devez suivre pour définir vos identifiants sont les suivantes :

- Un identifiant peut être constitué des lettres de l'alphabet de `a` à `z` et de `A` à `Z`, une lettre en majuscule étant différente d'une lettre en minuscule.
- Un identifiant peut être constitué de chiffres.
- Un identifiant peut contenir le caractère _.
- Un identifiant ne doit pas contenir les caractères suivants : un espace, une tabulation, un code de retour chariot, #, %, &...
- Un identifiant peut être en théorie de taille illimitée, bien qu'en pratique les compilateurs peuvent imposer des limites (larges).

Annexe B

Types de base du langage C++

Dans cette annexe nous présentons les types de base utilisables en langage C++ pour définir des variables ou des objets plus complexes.



On notera que pour les entiers signés la mention du mot-clef `signed` n'est pas obligatoire : par défaut un type entier est défini en signé. Nous présentons dans le tableau suivant l'intervalle de variation de ces types, en supposant que nous travaillons sur un ordinateur 32 bits (notamment les types `int` dépendent du type de processeur).

Type	**Taille en octets**	**Intervalle**
`bool`	1	`true` **ou** `false`
`char`	1	-128 à 127
`unsigned char`	1	0 **à** 255
`short`	2	-32768 **à** 32767
`unsigned short`	2	0 **à** 65535
`int`	4	-2147483648 **à** 2147483647
`unsigned int`	4	0 **à** 4294967295
`long`	8	-9223372036854775808 **à** 9223372036854775807
`unsigned long`	8	0 **à** 18446744073709551615

Type	**Intervalle**
`float`	+/- 1.2×10^{-38} **à** +/- $3.4 \times 10^{+38}$
`double`	+/- 2.2×10^{-308} **à** +/- $1.8 \times 10^{+308}$
`long double`	+/- 3.3×10^{-4932} **à** +/- $1.2 \times 10^{+4932}$

On notera que le codage des flottants n'est pas imposé par la norme ANSI du langage C++ ce qui implique que ces intervalles peuvent varier d'un compilateur à l'autre. De même, le type `long double` peut très bien être identique au type `double`.

Annexe **C**

Passage d'arguments à une fonction

C.1 LE PASSAGE PAR VALEUR

Dans cette section nous allons rappeler comment fonctionne le passage d'arguments par valeur à une fonction. Reprenons l'exemple donné dans la section 2.3 et illustré dans la figure C.1.

```
1  void f(int iP1, int * piP2, int & iP3)
2  {
3          ...
4          iP1=10;
5          ...
6  }
7
8  void main()
9  {
10  int iV1=5,iV2=5,iV3=5;
11  f(iV1,&iV2,iV3);
12 }
```

Lors de l'appel à une fonction `f` contenant un ou plusieurs arguments passés par valeur (dans l'exemple, l'argument `iV1`), le compilateur va réaliser la création d'un objet temporaire qui contiendra une copie de la variable passée par valeur en paramètre (action (2) dans la figure C.1). L'adresse de cet objet temporaire sera transmise

à la fonction (action (3) dans la figure C.1). Ainsi, la fonction appelée travaillera sur une copie de la variable initiale ce qui implique qu'après l'appel à la fonction `f` dans notre exemple (ligne 12), la variable `iV1` ne vaudra pas 10 mais 5. C'est la copie qui aura été modifiée dans la fonction `f`. Un inconvénient majeur du passage d'arguments par valeur est qu'il met en œuvre de la création d'objet temporaires initialisés par recopie de la variable de départ. Lorsque cette variable est d'un type de base du langage (`int`, `float`...) la copie est rapide, mais lorsqu'elle est d'une classe ou d'une structure créée par le programmeur, cette copie peut devenir plus volumineuse et coûteuse en temps.

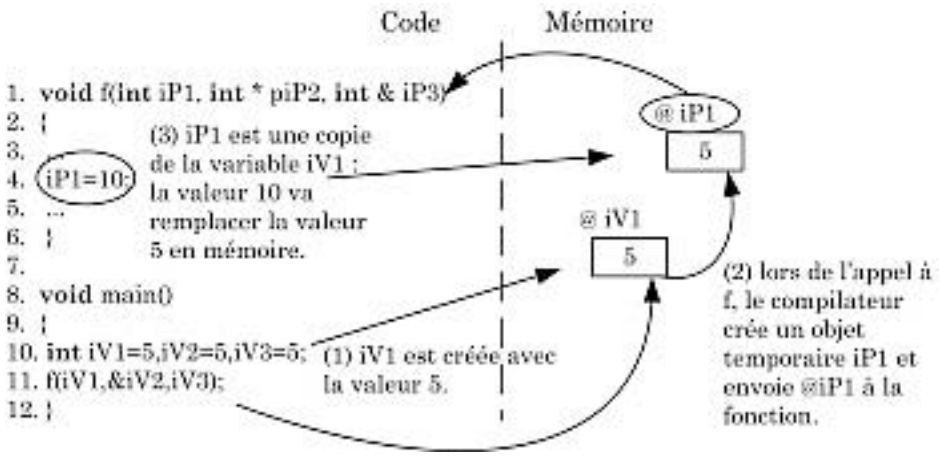

Code
Mémoire
1. void f(int iP1, int * piP2, int & iP3)
2. {
3.
4. iP1=10;
5. ...
6. }
7.
8. void main()
9. {
10. int iV1=5,iV2=5,iV3=5;
11. f(iV1,&iV2,iV3);
12. }
(3) iP1 est une copie de la variable iV1 ; la valeur 10 va remplacer la valeur 5 en mémoire.
(1) iV1 est créée avec la valeur 5.
@ iP1
5
@ iV1
5
(2) lors de l'appel à f, le compilateur crée un objet temporaire iP1 et envoie @iP1 à la fonction.

FIG. C.1: Le mécanisme du passage par valeur d'arguments à une fonction

Pour terminer, rappelons que pour une variable `iV1` :

– `iV1` réfère à la valeur de la variable,

– `&iV1` réfère à l'adresse de la variable (ce qui est noté *@iV1* dans la figure C.1), c'est-à-dire l'adresse en mémoire qui contient la valeur de la variable.

C.2 LE PASSAGE PAR ADRESSE

Dans cette section nous allons rappeler comment fonctionne le passage d'arguments par adresse à une fonction. Reprenons l'exemple donné dans la section 2.3 et illustré dans la figure C.2.

```
1  void f(int iP1, int * piP2, int & iP3)
2  {
3          ...
4          *iP2=10;
5          ...
6  }
7
8  void main()
9  {
10  int iV1=5,iV2=5,iV3=5;
11  f(iV1,&iV2,iV3);
12 }
```

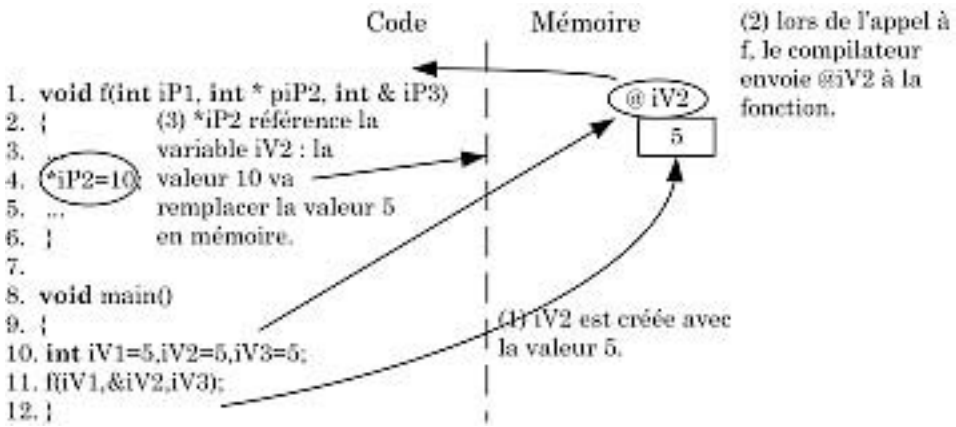

Code
Mémoire
(2) lors de l'appel à f, le compilateur envoie @iV2 à la fonction.
@ iV2
5
1. void f(int iP1, int * piP2, int & iP3)
2. {
3. ...
4. *iP2=10;
5. ...
6. }
7.
8. void main()
9. {
10. int iV1=5,iV2=5,iV3=5;
11. f(iV1,&iV2,iV3);
12. }
(3) *iP2 référence la variable iV2 : la valeur 10 va remplacer la valeur 5 en mémoire.
(1) iV2 est créée avec la valeur 5.

FIG. C.2: Le mécanisme du passage par adresse d'arguments à une fonction

Le passage d'un argument par adresse à une fonction `f` se fait simplement en passant une adresse, soit en passant un pointeur lors de l'appel à la fonction, soit en passant l'adresse de la variable (comme c'est le cas de la variable `iV2`, ligne 11). L'adresse étant transmise à la fonction `f`, celle-ci référence directement la variable d'origine moyennant un niveau d'indirection supplémentaire, c'est-à-dire que pour une adresse `iP2` :

- `iP2` réfère à l'adresse de la variable (ce qui est noté *@iV2* dans la figure C.2) c'est-à-dire l'adresse en mémoire qui contient la valeur de la variable,
- `*iP2` réfère à la valeur de la variable ,
- `&iP2` réfère à l'adresse de l'adresse `iP2`, c'est-à-dire l'adresse de la zone mémoire qui contient l'adresse `iP2`.

Annexe D

Liste des opérateurs surchargeables

Les opérateurs suivants sont les opérateurs du langage C++ qui peuvent être surchargés. Pour chaque opérateur est indiquée sa priorité par rapport aux règles d'associativité. Un opérateur de priorité 1 sera évalué avant un autre opérateur ayant, par exemple, la priorité 5 : plus la valeur de la priorité est faible et plus l'opérateur est prioritaire.

Opérateur	Priorité	
`->`	1	Accès indirect à un membre
`[]`	1	Accès au membre d'un tableau
`()`	1	Appel d'une fonction
`++`	1	Incrémentation postfixée
`--`	1	Décrémentation postfixée
`+`	2	Plus unaire
`-`	2	Moins unaire
`++`	2	Incrémentation préfixée
`--`	2	Décrémentation préfixée
`!`	2	Négation logique
`~`	2	Complément des bits
`&`	2	Opérateur d'adressage
`*`	2	Valeur de
`(type)`	2	Conversion explicite (cast) où `type` est à remplacer par un type de base du langage
`new`	2	Allocation mémoire d'un pointeur
`delete`	2	Désallocation mémoire d'un pointeur
`static_cast`	2	Conversion explicite vérifiée à la compilation
`dynamic_cast`	2	Conversion explicite vérifiée à l'exécution

Opérateur	Priorité	
`->*`	3	Accès indirect à la valeur d'un attribut dynamique
`*`	4	Multiplication
`/`	4	Division
`%`	4	Modulo
`+`	5	Addition arithmétique
`-`	5	Soustraction arithmétique
`«`	6	Décalage à gauche de bits
`»`	6	Décalage à droite de bits
`<`	7	Inférieur à
`>`	7	Supérieur à
`<=`	7	Inférieur ou égal à
`>=`	7	Supérieur ou égal à
`==`	8	Égale à
`!=`	8	Différent de
`&`	9	ET bit à bit
`^`	10	OU exclusif bit à bit
`|`	11	OU bit à bit
`&&`	12	ET logique
`||`	13	OU logique
`=`	14	Affectation
`*=`	15	Multiplication avant affectation
`/=`	15	Division avant affectation
`%=`	15	Modulo avant affectation
`+=`	15	Addition avant affectation
`-=`	15	Soustraction avant affectation
`&=`	15	ET bit à bit avant affectation
`≙`	15	OU exclusif bit à bit avant affectation
`|=`	15	OU bit à bit avant affectation
`«=`	15	Décalage à gauche avant affectation
`»=`	15	Décalage à droite avant affectation
`throw`	16	Levée d'exception

Annexe E

La classe `Cexception`

Voici un exemple de classe pour la gestion des exceptions telle que nous l'avons présentée dans cet ouvrage. La classe `Cexception` peut naturellement être enrichie selon l'utilisation que vous en faites.

Nous présentons tout d'abord l'interface de cette classe (qui se trouve dans le fichier `Cexception.h`) avant d'en présenter son corps (qui se trouve dans le fichier `Cexception.cpp`). Cette classe a été compilée avec MS Visual C++ 6.0.

Fichier Cexception.h

```
1  /*************************************************************
2  Titre : Classe pour la gestion des exceptions
3  *************************************************************
4  Auteur    : V. T'kindt
5  Version : 1.0
6  Date      : 10/07/2006
7  ——————————-
8  Lecteur : V. T'kindt
9  Date      : 11/07/2006
10 *************************************************************
11 Auteur    :
12 Version   :
13 Date      :
14 ——————————-
15 Lecteur :
16 Date      :
17 *************************************************************
18 Interface de la classe Cexception.
19 Cette classe permet de créer des objets qui sont remontés
20 lors de la levée d'exceptions dans votre programme.
21 *************************************************************/
22
23 // Définition de la valeur initiale d'une exception
24 #define FAUX 0
25
26 class Cexception
27 {
28  // Cette classe représente une ou plusieurs exceptions levées
29  // par le programme
30
31  //Attributs :
32  private :
33       unsigned int uiEXCvaleur ; //Cette variable contient la valeur de l'exception
34
35  /*ETAT INITIAL
36      uiEXCvaleur = FAUX*/
37
38  //Primitives :
39  public :
40    Cexception() ;
41    /* Constructeur par défaut de la classe
42          E : néant
43          nécessite : néant
```

```
44          S : néant
45          entraîne : l'exception est initialisé à FAUX */
46      ~Cexception(){}
47      /* Destructeur de la classe
48          E : néant
49          nécessite : néant
50          S : néant
51          entraîne : l'exception est détruite*/
52      void EXCmodifier_valeur(unsigned int);
53      /* Cette fonction permet de modifier la valeur de l'exception
54          E : nouvelle valeur
55          nécessite : néant
56          S : néant
57          entraîne : la valeur de l'exception est modifiée*/
58      unsigned int EXClire_valeur();
59      /* Cette fonction permet de consulter la valeur de l'exception
60          E : néant
61          nécessite : néant
62          S : valeur de l'exception
63          entraîne : la valeur de l'exception est retournée*/
64  };
```

Fichier Cexception.cpp

```
1  /*************************************************************
2  Titre : Classe pour la gestion des exceptions
3  *************************************************************
4  Auteur   : V. T'kindt
5  Version : 1.0
6  Date     : 10/07/2006
7  ——————————-
8  Lecteur : V. T'kindt
9  Date     : 11/07/2006
10 *************************************************************
11 Auteur   :
12 Version  :
13 Date     :
14 ——————————-
15 Lecteur :
16 Date     :
17 *************************************************************
18 Corps de la classe Cexception.
19 Cette classe permet de créer des objets qui sont remontés
20 lors de la levée d'exceptions dans votre programme.
21 *************************************************************/
22
23
24 /*CLASSE Cexception
25   DOCUMENTATION
26         Attributs : uiEXCvaleur, entier, contient la valeur de l'exception
27         Structure : Cette classe contient une méthode de modification et une méthode
28                           de consultation de la valeur de l'exception
29         Méthode : néant
30         Modules internes :*/
31                 #include <fstream.h>
32                 #include "Cexception.h"
33
34 //CORPS
35
36 /*************************************************************
37 Nom : Cexception
38 *************************************************************
39 Constructeur par défaut de la classe Cexception : permet
40 d'initialiser un objet
41 *************************************************************
42 Entrée : rien
```

```
43 Nécessite : néant
44 Sortie : rien
45 Entraîne : L'exception est intialisée à FAUX
46 ************************************************************/
47 Cexception : :Cexception()
48 {
49  uiEXCvaleur = FAUX;
50  // l'exception est initialisée
51 }
52
53 /************************************************************
54 Nom : EXCmodifier_valeur
55 ************************************************************
56 Cette fonction permet de modifier la valeur de l'exception
57 ************************************************************
58 Entrée : la nouvelle valeur de l'exception
59 Nécessite : néant
60 Sortie : rien
61 Entraîne : L'exception est modifiée
62 ************************************************************/
63 void Cexception : :EXCmodifier_valeur(unsigned int val)
64 {
65  uiEXCvaleur = val;
66  // l'exception est modifiée
67 }
68
69 /************************************************************
70 Nom : EXClire_valeur
71 ************************************************************
72 Cette fonction permet de consulter la valeur de l'exception
73 ************************************************************
74 Entrée : rien
75 Nécessite : néant
76 Sortie : la valeur de l'exception
77 Entraîne : L'exception est retournée
78 ************************************************************/
79 inline unsigned int Cexception : :EXClire_valeur()
80 {
81  return(uiEXCvaleur);
82 }
```

Annexe F

La classe `Cliste`

Voici un exemple de classe pour la gestion d'une liste d'éléments dont le type peut être fixé à la compilation de façon centralisée dans l'interface. Cette classe nécessite la classe `Cexception` vue dans l'annexe E. La classe `Cliste` peut naturellement être étendue selon l'utilisation que vous en faites.

Nous présentons tout d'abord l'interface de cette classe (qui se trouve dans le fichier `Cliste.h`) avant d'en présenter son corps (qui se trouve dans le fichier `Cliste.cpp`). Cette classe a été compilée avec MS Visual C++ 6.0.

Fichier Cliste.h

```
1  /*************************************************************
2  Titre : Classe pour la gestion de liste d'éléments d'un type de base
3  *************************************************************
4  Auteur   : V. T'kindt
5  Version : 1.0
6  Date     : 10/07/2006
7  ——————————-
8  Lecteur : V. T'kindt
9  Date     : 11/07/2006
10 *************************************************************
11 Auteur   :
12 Version  :
13 Date     :
14 ——————————-
15 Lecteur :
16 Date     :
17 *************************************************************
18 Interface de la classe Cliste.
19 Cette classe permet de créer des listes d'éléments dont le type peut être fixé
20 de façon centralisée.
21 *************************************************************/
22
23 // Définition des valeurs des exceptions pouvant être levées dans cette classe
24
25 #define Ajout_impossible 101
26 #define Suppression_impossible 102
27 #define Position_hors_liste 103
28
29 // Définition du type d'un élément de la liste
30 // Il faut modifier le type de base ci-dessous pour changer
31 // le type des éléments de la liste
32 typedef int Telement;
33
34 class Cliste
35 {
36  // Cette classe représente une ou plusieurs listes d'éléments
37
38  // Attributs :
39  private :
40   unsigned int uiLIStaille; //Cet attribut contient le nombre d'éléments de la liste
41   Telement * pTeLISliste; // Cet attribut contient la liste des éléments
42
43  /* ETAT INITIAL
```

```
44        uiLIStaille = 0
45        pTeLISliste = NULL */
46
47    // Primitives :
48    public :
49      Cliste() ;
50      /* Constructeur par défaut de la classe
51            E : néant
52            nécessite : néant
53            S : néant
54            entraîne : la liste est initialisée à vide */
55      Cliste(const Cliste &) ;
56      /* Constructeur de recopie de la classe
57            E : l'objet que l'on recopie dans l'objet en cours
58            nécessite : néant
59            S : néant
60            entraîne : L'objet en cours contient une copie de l'objet passé en paramètre */
61      ~Cliste() ;
62      /* Destructeur de la classe
63            E : néant
64            nécessite : néant
65            S : néant
66            entraîne : l'exception est détruite*/
67      Cliste & operator=(const Cliste &) ;
68      /* Surcharge de l'opérateur d'affectation
69            E : l'objet que l'on affecte dans l'objet en cours
70            nécessite : néant
71            S : L'objet en cours qui a été recopié
72            entraîne : (L'objet en cours contient une copie de l'objet passé en paramètre) ou
73                  (Exception Ajout_impossible : pas assez de mémoire libre) */
74      void LISajouter_element(Telement elem, unsigned int pos) ;
75      /* Cette fonction permet d'ajouter un élément dans la liste à la position spécifiée
76            E : elem, la valeur à ajouter
77                pos, la position d'insertion
78            nécessite : néant
79            S : néant
80            entraîne : (l'élément est ajouté à la position requise) ou
81                  (Exception Ajout_impossible : pas assez de mémoire libre) ou
82                  (Exception Position_hors_liste : pos>taille) */
83      unsigned int LISlire_taille() ;
84      /* Cette fonction permet de connaître la taille de la liste
85            E : rien
86            nécessite : néant
87            S : la taille de la liste
88            entraîne : (la taille de la liste est retournée) */
89      Telement LISlire_element(unsigned int pos) ;
```

```
90      /* Cette fonction permet de lire la valeur d'un élément dans la liste
91           E : la position de l'élément à lire
92           nécessite : néant
93           S : la valeur de l'élément
94           entraîne : (la valeur de l'élément est retournée) ou
95                      (Exception Position_hors_liste : pos>=taille) */
96      void LISmodifier_element(Telement elem, unsigned int pos);
97      /* Cette fonction permet de modifier la valeur d'un élément dans la liste
98           E : elem, la nouvelle valeur
99                pos, la position de l'élément à modifier
100          nécessite : néant
101          S : néant
102          entraîne : (L'élément est modifié) ou
103                     (Exception Position_hors_liste : pos>=taille) */
104     Telement & operator[](unsigned int pos);
105     /* Cette fonction permet d'accéder à la valeur d'un élément dans la liste
106          E : la position de l'élément à lire
107          nécessite : néant
108          S : l'élément
109          entraîne : (l'élément est accessible) ou
110                     (Exception Position_hors_liste : pos>=taille) */
111     void LISsupprimer_element(unsigned int pos);
112     /* Cette fonction permet de supprimer un élément dans la liste à la position spécifiée
113          E : pos, la position de suppression
114          nécessite : néant
115          S : néant
116          entraîne : (l'élément est supprimé à la position requise) ou
117                     (Exception Suppression_impossible : taille=0) ou
118                     (Exception Position_hors_liste : pos>=taille) */
119 };
```

Fichier Cliste.cpp

```
1  /*************************************************************
2  Titre : Classe pour la gestion de liste d'éléments d'un type de base
3  *************************************************************
4  Auteur    : V. T'kindt
5  Version : 1.0
6  Date      : 10/07/2006
7  ——————————-
8  Lecteur : V. T'kindt
9  Date      : 11/07/2006
10 *************************************************************
11 Auteur    :
12 Version   :
13 Date      :
14 ——————————-
15 Lecteur :
16 Date      :
17 *************************************************************
18 Corps de la classe Cliste.
19 Cette classe permet de créer des listes d'éléments dont le type peut
20 être fixé de façon centralisée.
21 *************************************************************/
22
23 /*CLASSE Cliste
24   DOCUMENTATION
25         Attributs : uiLIStaille, entier, contient le nombre d'éléments dans la liste
26                                 pTeliste, pointeur, contient les éléments de la liste
27         Structure : Cette classe contient des méthodes permettant l'ajout, la suppression,
28                     la modification et la consultation d'éléments dans la liste.
29         Méthode : L'allocation de la liste est gérée dynamiquement par pointeur
30         Modules internes :*/
31                 #include <fstream.h>
32                 #include <malloc.h>
33                 #include "Cliste.h"
34                 #include "Cexception.h"
35
36 //CORPS
37
38 /*************************************************************
39 Nom : Cliste
40 *************************************************************
41 Constructeur par défaut de la classe Cliste : permet d'initialiser un objet
42 *************************************************************
43 Entrée : rien
```

```
44 Nécessite : néant
45 Sortie : rien
46 Entraîne : la liste est initialisée à vide
47 ****************************************************************/
48 Cliste : :Cliste()
49 {
50  uiLIStaille = 0;
51  pTeLISliste = NULL;
52  // la liste est vide
53 }
54
55 /****************************************************************
56 Nom : ~Cliste
57 ****************************************************************
58 Destructeur de la classe Cliste : permet de détruire un objet
59 ****************************************************************
60 Entrée : rien
61 Nécessite : néant
62 Sortie : rien
63 Entraîne : la liste est désallouée
64 ****************************************************************/
65 Cliste : :~Cliste()
66 {
67  free(pTeLISliste);
68  // la liste est désallouée
69 }
70
71 /****************************************************************
72 Nom : Cliste (R)
73 ****************************************************************
74 Constructeur de recopie de la classe Cliste : permet
75 d'initialiser un objet par recopie à partir d'un autre
76 ****************************************************************
77 Entrée : l'objet que l'on recopie dans l'objet en cours
78 Nécessite : néant
79 Sortie : rien
80 Entraîne : L'objet en cours contient une copie de l'objet passé en paramètre
81 ****************************************************************/
82 Cliste : :Cliste(const Cliste & objet)
83 {
84  int iBoucle;
85
86  uiLIStaille = objet.uiLIStaille;
87  pTeLISliste = (Telement *) malloc(uiLIStaille*sizeof(Telement));
88  // pTeliste pointe sur une zone mémoire allouée mais "vide"
89
```

```
90   for (iBoucle=0;iBoucle<uiLIStaille; iBoucle++)
91             pTeLISliste[iBoucle]=objet.pTeLISliste[iBoucle];
92   // la liste objet est recopiée dans l'objet en cours
93  }
94
95  /*************************************************************
96  Nom : operator=
97  *************************************************************
98  Surcharge de l'opérateur d'affectation
99  *************************************************************
100 Entrée : l'objet que l'on affecte dans l'objet en cours
101 Nécessite : néant
102 Sortie : L'objet en cours qui a été recopié
103 Entraîne : (L'objet en cours contient une copie de l'objet passé en paramètre) ou
104                    (Exception Ajout_impossible : pas assez de mémoire libre)
105 *************************************************************/
106 Cliste & Cliste : :operator=(const Cliste & objet)
107 {
108  int iBoucle;
109
110  if (uiLIStaille!=0)
111  { // Il faut vider la liste en cours
112    uiLIStaille=0;
113    free(pTeLISliste);
114  }
115
116  uiLIStaille=objet.uiLIStaille;
117  pTeLISliste = (Telement *) malloc(uiLIStaille*sizeof(Telement));
118  if (pTeLISliste==NULL)
119  { // L'agrandissement de la liste est impossible : levée d'une exception
120    Cexception pasmemoire;
121    pasmemoire.EXCmodifier_valeur(Ajout_impossible);
122    throw(pasmemoire);
123  }
124  // pTeliste pointe sur une zone mémoire allouée mais "vide"
125
126  for (iBoucle=0;iBoucle<uiLIStaille; iBoucle++)
127            pTeLISliste[iBoucle]=objet.pTeLISliste[iBoucle];
128  // la liste objet est recopiée dans l'objet en cours
129  return *this;
130 }
131
132 /*************************************************************
133 Nom : LISajouter_element
134 *************************************************************
135 Cette fonction permet d'ajouter un élément dans la liste à la position spécifiée
```

```
136 ************************************************************
137 Entrée : elem, la valeur à ajouter
138               pos, la position d'insertion
139 Nécessite : néant
140 Sortie : rien
141 Entraîne : (l'élément est ajouté à la position requise) ou
142                   (Exception Ajout_impossible : pas assez de mémoire libre) ou
143                   (Exception Position_hors_liste : pos>taille)
144 ************************************************************/
145 void Cliste : :LISajouter_element(Telement elem, unsigned int pos)
146 {
147  Telement * pTetmp;
148  int iBoucle;
149
150  if (pos>uiLIStaille)
151  { // L'insertion a lieu hors tableau : levée d'une exception
152   Cexception horstableau;
153   horstableau.EXCmodifier_valeur(Position_hors_liste);
154   throw(horstableau);
155  }
156
157  pTetmp=(Telement *)realloc(pTeLISliste,(uiLIStaille+1)*sizeof(Telement));
158  if (pTetmp==NULL)
159  { // L'agrandissement de la liste est impossible : levée d'une exception
160   Cexception pasmemoire;
161   pasmemoire.EXCmodifier_valeur(Ajout_impossible);
162   throw(pasmemoire);
163  }
164
165  uiLIStaille++;
166  pTeLISliste=pTetmp;
167  for (iBoucle=uiLIStaille;iBoucle>pos;iBoucle--)
168            pTeLISliste[iBoucle]=pTeLISliste[iBoucle-1];
169  pTeLISliste[pos]=elem;
170  // L'élément est inséré dans la liste à la position demandée
171 }
172
173 /************************************************************
174 Nom : LISlire_element
175 ************************************************************
176 Cette fonction permet de connaître la taille de la liste
177 ************************************************************
178 Entrée : rien
179 Nécessite : néant
180 Sortie : la taille de la liste
181 Entraîne : (la taille de la liste est retournée)
```

```
182 ***************************************************************/
183 unsigned int Cliste : :LISlire_taille()
184 {
185  return (uiLISTaille);
186 }
187
188 /***************************************************************
189 Nom : LISlire_element
190 ***************************************************************
191 Cette fonction permet de lire la valeur d'un élément dans la liste
192 ***************************************************************
193 Entrée : la position de l'élément à lire
194 Nécessite : néant
195 Sortie : la valeur de l'élément
196 Entraîne : (la valeur de l'élément est retournée) ou
197                    (Exception Position_hors_liste : pos>=taille)
198 ***************************************************************/
199 Telement Cliste : :LISlire_element(unsigned int pos)
200 {
201  if (pos>=uiLISTaille)
202  { // L'insertion a lieu hors tableau : levée d'une exception
203    Cexception horstableau;
204    horstableau.EXCmodifier_valeur(Position_hors_liste);
205    throw(horstableau);
206  }
207
208  // La position est dans le tableau : on retourne l'élément demandé
209  return pTeLISliste[pos];
210 }
211
212 /***************************************************************
213 Nom : LISmodifier_element
214 ***************************************************************
215 Cette fonction permet de modifier la valeur d'un élément dans la liste
216 ***************************************************************
217 Entrée : elem, la nouvelle valeur
218             pos, la position de l'élément à modifier
219 Nécessite : néant
220 Sortie : rien
221 Entraîne : (L'élément est modifié) ou
222                (Exception Position_hors_liste : pos>=taille)
223 ***************************************************************/
224 void Cliste : :LISmodifier_element(Telement elem, unsigned int pos)
225 {
226  if (pos>=uiLISTaille)
227  { // L'insertion a lieu hors tableau : levée d'une exception
```

```
228     Cexception horstableau ;
229     horstableau.EXCmodifier_valeur(Position_hors_liste) ;
230     throw(horstableau) ;
231    }
232
233    // La position est dans le tableau : on modifie l'élément demandé
234    pTeLISliste[pos]=elem ;
235   }
236
237   /*************************************************************
238   Nom : operator[ ]
239   *************************************************************
240   Cette fonction permet d'accéder à la valeur d'un élément dans la liste
241   *************************************************************
242   Entrée : la position de l'élément à lire
243   Nécessite : néant
244   Sortie : l'élément
245   Entraîne : (L'élément est accessible) ou
246                       (Exception Position_hors_liste : pos>=taille)
247   *************************************************************/
248   Telement & Cliste : :operator[ ](unsigned int pos)
249   {
250    if (pos>=uiLIStaille)
251    { // L'insertion a lieu hors tableau : levée d'une exception
252     Cexception horstableau ;
253     horstableau.EXCmodifier_valeur(Position_hors_liste) ;
254     throw(horstableau) ;
255    }
256
257    // La position est dans le tableau : on accède à l'élément demandé
258    return(pTeLISliste[pos]) ;
259   }
260
261   /*************************************************************
262   Nom : LISsupprimer_element
263   *************************************************************
264   Cette fonction permet de supprimer un élément dans la liste à la position spécifiée
265   *************************************************************
266   Entrée : la position de l'élément à supprimer
267   Nécessite : néant
268   Sortie : rien
269   Entraîne : (l'élément est supprimé à la position requise) ou
270                       (Exception Position_hors_liste : pos>=taille)
271   *************************************************************/
272   void Cliste : :LISsupprimer_element(unsigned int pos)
273   {
```

```
274   int iBoucle;
275
276   if (pos>=uiLIStaille)
277   { // L'insertion a lieu hors tableau : levée d'une exception
278     Cexception horstableau;
279     horstableau.EXCmodifier_valeur(Position_hors_liste);
280     throw(horstableau);
281   }
282
283   for (iBoucle=pos;iBoucle<uiLIStaille-1;iBoucle++)
284             pTeLISliste[iBoucle]=pTeLISliste[iBoucle+1];
285   uiLIStaille--;
286
287   // L'élément est supprimé
288 }
```

Bibliographie

[1] E.W. Dijkstra. *A discipline of Programming*. Prentice-Hall, 1976.

[2] D. Gustafson. *Génie logiciel*. Ediscience, 2003.

[3] I. Horton. *ANSI C++. The complete language.* Apress, 2004.

[4] B. Meyer. *Object-oriented software construction*. 2nd edition, Prentice-Hall, 1997.

[5] D.L. Parnas. On the criteria to be used in decomposing systems into modules. *Communications of the ACM*, 15(12), 1972.

[6] D.L. Parnas. Software engineering principles. *Informatic*, 22(4), 1984.

[7] G. Pierra. *Les bases de la programmation et du génie logiciel*. Dunod informatique, 1991.

[8] J. Printz. *Le génie logiciel*. PUF, 2005.

[9] J. Printz. *Architecture logicielle*. InfoPro, Dunod, 2006.

Index

`#ifndef`, 32
`abort`, 88
`catch`, 81
`cerr`, 161
`cin`, 161, 164
`class`, 39, 136, 141
`clog`, 161
`const`, 29, 52, 61, 71, 72
`cout`, 161, 162
`delete`, 57, 60, 62, 124
`dynamic_cast`, 127
`endl`, 164
`explicit`, 131
`export`, 142
`fail`, 166
`flush`, 164
`free`, 57
`friend`, 76, 77
`fstream`, 165
`ifstream`, 165
`inline`, 67, 141
`is_open`, 166
`malloc`, 57
`new`, 57, 62, 124
`ofstream`, 165, 167
`operator`, 115
`private`, 39
`protected`, 39
`public`, 39
`seekg`, 168
`seekp`, 168
`static_cast`, 127
`static`, 27, 46, 47, 61, 70
`struct`, 37
`tellg`, 168
`tellp`, 168
`template`, 136, 141
`terminate`, 88
`this`, 71
`throw`, 80
`try`, 81
`unexpected`, 88
`using`, 95
`virtual`, 104, 151
`free`, 60
`malloc`, 57
`new`, 57

Accesseur, 16, 18, 40, 69
Allocation, 57
Attribut statique, 47
Auto-référence, 71

Cast, 126
CDU, 128
Chaîne de conversions, 128, 132
Classe, 38
  Abstraite, 157
  Déclaration, 38
  Utilisation, 41
Commentaire, 5, 23
  De fin de ligne, 24
Compatibilité ascendante, 16, 40
Constructeur, 49, 55, 98, 105, 125, 129
  De recopie, 45, 51, 54, 58, 62, 100
  De recopie par défaut, 53
  Par défaut, 50, 51, 54, 57, 58, 63
  Par défaut par défaut, 50
Continuité modulaire, 12
Contrôle d'accès, 39, 40
Conversion
  Définie par l'utilisation, 128
  Explicite, 126, 131
  Implicite, 126, 131
Corps, 11
Cycle de vie, 2

Dérivation, 91
Désallocation, 60
Destructeur, 49, 56, 98, 105, 125
Dualité précondition/postcondition, 10

Encapsulation, 11
  Des données, 14, 35, 37, 40, 75
  Des modifications, 12
Entrée/Sortie, 159
Exception, 5, 10, 19, 79, 90, 166, 170
  Valide, 87

Fichier, 165
  Accès direct, 168

Généricité, 135, 149
Génie logiciel, 1
Gestion
  Hiérarchisée des exceptions, 85
  Mémoire, 57
Gestionnaire d'exceptions, 83

Héritage
  Accès externe, 94
  Accès interne, 94
  Statut dans la classe fille, 94
Héritage multiple, 101
  Constructeur, 105
  Destructeur, 105
  Duplication par héritage, 104
  Membres homonymes, 101
Héritage simple, 91
  Constructeur, 98
  Constructeur de recopie, 100
  Contrôle d'accès, 93
  Destructeur, 98

Inclusion d'interfaces, 31
Initialisateur, 61
Initialisation
  D'attributs, 55
  Lors d'une déclaration, 47, 49
Interface, 11

Linkage
  Dynamique, 150, 154, 155
  Statique, 150, 155

Méthode
  Virtuelle, 151
  Virtuelle pure, 156
Manipulateur, 171
Masque, 167–169
Mode
  Binaire, 160
  Texte, 160, 167
Mot-clefs, 211
Mots-clefs, 23

Nom long, 36, 48, 96
Nommage, 20
Normes de rédaction, 20, 21

Objet
  Automatique, 62, 67
  Dynamique, 62
  Statique, 61
  Temporaire, 62, 130
Occultation, 96, 114
Opérateur
  D'accès, 121
  D'affectation, 45, 118
  D'extraction, 161
  D'indexation, 120
  D'insertion, 161
  De conversions, 128
  De déréférencement, 123
  De résolution de portée, 36, 48, 70
  Surchargeable, 219
Orthogonalité, 19

Paramètre option, 15
Passage d'arguments
  Par adresse, 28, 67, 216
  Par défaut, 65
  Par référence, 28, 52, 67
  Par valeur, 28, 52, 62, 67, 215
Patron de classes, 140
  Classe amie, 145
  Fonction amie, 145
  Instanciation, 143
  Spécialisation, 146
  Spécialisation partielle, 147
  Spécialisation totale, 147
  Type par défaut, 144
Patron de fonctions, 136, 142
  Instanciation, 138
  Spécialisation, 139
  Surcharge, 139
Pointeur de fonction, 73
Polymorphisme, 149, 150
Postcondition, 9, 80
Précondition, 9
Protection modulaire, 18, 40

Référence, 28, 30
Réutilisation, 17

Recopie
  D'objets, 29
  Membre à membre, 38, 42, 53

Séparation des difficultés, 14
Spécification logique, 8, 19, 21
Structure, 35
  Déclaration, 35
  Définition, 36
  Utilisation, 37
Surcharge, 107
  D'opérateur, 114
    Ordre des opérandes, 116
  De type, 126
Surcharge de fonction
  Conversion standard, 109
  Correspondance exacte, 108
  Dominance, 112, 113
  Promotion numérique, 109

Tableau d'éléments, 57
Terminaison du programme, 88
Type, 213
  De retour, 31

Variable
  Automatique, 27, 60
  Déclaration, 25
  Gestion, 24
  Globale, 27
  Locale, 27
  Portée, 25, 26
  Référence, 30
  Statique, 46, 60

SCIENCES SUP

Vincent T'kindt

PROGRAMMATION EN C++ ET GÉNIE LOGICIEL

L'apprentissage de la **programmation en langage C++** recouvre deux éléments essentiels : l'un est lié à l'apprentissage du langage en lui-même (les instructions et les règles) et l'autre, le **génie logiciel**, est lié à la façon d'écrire ces instructions pour limiter le nombre de bugs et favoriser un développement propre et efficace du programme.

Dans cet ouvrage, vous trouverez tous les éléments du langage mais également un **ensemble de recommandations** qui vous guideront dans l'écriture de vos programmes :

- La première partie est consacrée à la présentation des notions élémentaires du génie logiciel, ces notions étant illustrées ensuite dans le reste de l'ouvrage.
- La deuxième partie est dédiée à l'apprentissage des éléments de base du langage C++. À partir de là, vous saurez créer des classes, y définir des attributs et des méthodes.
- La troisième partie présente les mécanismes particuliers du langage : l'amitié, les exceptions, l'héritage, la surcharge, les patrons, le polymorphisme et les flots.

De **nombreux exemples** ponctuent l'apprentissage des notions du cours et des **exercices corrigés** permettent de confirmer les automatismes acquis. Le **code source** des exemples est téléchargeable.

VINCENT T'KINDT

est maître de conférences au département Informatique de l'École Polytechnique de l'université François Rabelais de Tours. Il y enseigne notamment depuis plus de dix ans les cours de génie logiciel et langage C++.

MATHÉMATIQUES
PHYSIQUE
CHIMIE
SCIENCES DE L'INGÉNIEUR
INFORMATIQUE
SCIENCES DE LA VIE
SCIENCES DE LA TERRE

9 782100 506347

6494009
ISBN 978-2-10-050634-7

LICENCE 1 2 3 | MASTER 4 5 | DOCTORAT 6 7 8

COMPLÉMENTS en ligne

www.dunod.com

DUNOD